Konstrukcja kompilatorów

Projekt nr 4

Termin oddawania: 23 grudnia 2018

UWAGA! Przed przystąpieniem do zadania proszę zapoznać się ze stronami w systemie SKOS poświęconymi pracowni. Część ze stron została uaktualniona.

Czwartym projektem na pracownię z konstrukcji kompilatorów jest algorytm alokacji rejestrów. W systemie SKOS znajdują się nowe źródła kompilatora, gdzie w katalogu «`source/mod_student`» zamieściliśmy szablon rozwiązania w pliku «`regalloc.ml`». W katalogu «`tests/pracownia4`» znajdą Państwo przygotowany przez nas zestaw testów. Części algorytmu odpowiedzialne za złączanie-rejestrów, spilling oraz liczenie jego kosztów, są zaimplementowane przez nas. W przygotowanym przez nas szablonie znajduje się informacja jak ich użyć.

Schemat algorytmu Schemat algorytmu z lotu ptaka możemy opisać następującym pseudokodem:

```
1  InterferenceGraph BuildCoalesceLoop() {
2      InterferenceGraph infg;
3      bool program_code_changed;
4      do {
5          infg = Build();
6          program_code_changed = Coalesce(infg);
7      } while (program_code_changed);
8      return infg;
9  }
10
11 void MainLoop() {
12     bool spilling_executed;
13     do {
14         InterefrenceGraph infg = BuildCoalesceLoop();
15         SpillCosts scosts      = ComputeSpillCosts(infg);
16         SimplifyStack stack    = Simplify(scosts, infg);
17         RegisterList spill     = Select(stack, infg);
18         if (spill is empty) {
19             spilling_executed = false;
20         } else {
21             spilling_executed = true;
22             Spill(spill);
23         }
24     } while (spilling_executed);
25 }
26
27 RegisterMapping AllocateRegisters() {
28     MainLoop();
29     return BuildRegisterMapping();
30 }
```

Jest to schemat nieodbiegający od tego, który został przedstawiony Państwu na wykładzie. Procedura `BuildCoelesceLoop` realizuje schemat budowania grafu interferencji oraz złączania rejestrów. Proces jest powtarzany, dopóki dopóty da się złączyć jakieś rejestry (*register coalescing*). Następnie na podstawie grafu interferencji i kodu programu uruchamiana jest heurystyka wyliczająca koszta spillingu – za ten proces odpowiada na schemacie procedura `ComputeSpillCosts`. Następnie jest uruchamiana procedura `Simplify` odpowiedzialna za upraszczanie grafu. Ta procedura zwraca stos wierzchołków utworzony podczas tego procesu. Na szczycie stosu jest ten wierzchołek, który został usunięty ostatni. Na dnie stosu jest ten wierzchołek, który został usunięty jako pierwszy. Następnie jest uruchamiana procedura `Select` odpowiedzialna, za kolorowanie grafu. Zwraca ona listę rejestrów, których nie udało się pokolorować i trzeba zastosować na nich

spilling – za co odpowiada procedura `Spill`. Jeżeli lista rejestrów przeznaczonych do spillingu nie była pusta, powtarzany jest cały algorytm. W przeciwnym wypadku odczytujemy mapowanie rejestrów z pokolorowanego grafu.

Faza simplify

```
SimplifyStack Simplify(SpillCosts scosts, IntereferenceGraph infg) {
    SimplifyStack result;

    while (infg is not empty) {
        Vertex v = select from infg vertex with lowest degree;
        if (v is not REG_Tmp) return result;
        if (v.degree < N) {
            result.push(v);
            infg.remove(v);
        } else {
            v = SelectSpillCandidate(infg, scosts);
            result.push(v);
            infg.remove(v);
        }
    }

    return result;
}
```

Procedura `Simplify` w sposób iteracyjny kasuje wierzchołki z grafu, aż ten nie będzie posiadał żadnych wierzchołków reprezentujących rejestry tymczasowe. W naszym rozwiązaniu przyjęliśmy, że wierzchołki reprezentujące rejestry inne niż tymczasowe (`REG_Hard` oraz `REG_Spec`) mają stopień nieskończony (a w rzeczywistości `max_int`). Dzięki temu założeniu nasz algorytm zawsze wybiera rejestry tymczasowe jeżeli takie jeszcze są w grafie. Gdy wybrał natomiast rejestr nietymczasowy to wiedzieliśmy, że wszystkie rejestry tymczasowe zostały już odłożone na stos i można zakończyć procedurę.
Jeżeli wybrany wierzchołek ma stopień mniejszy niż liczba dostępnych rejestrów (kolorów) to kasujemy go po prostu z grafu; odkładamy na stos; kontynuujemy procedurę. W przeciwnym wypadku nie potrafimy zagwarantować, że wybrany wierzchołek będzie można pokolorować. Wybieramy więc nowy wierzchołek kierując się kosztem spillingu; kasujemy wybrany wierzchołek z grafu; odkładamy go na stos; kontynuujemy procedurę.
Procedura `SelectSpillCandidate` wybiera ten wierzchołek w grafie, który ma najmniejszy stosunek wyliczonego wcześniej kosztu spillingu do stopnia w obecnym grafie interferencji. Jest to popularna i prosta heurystyka.

Faza select

```
RegisterList Select(SimplifyStack stack, InterferenceGraph infg) {
    RegisterList actual_spills;
    while (stack is not empty) {
        Vertex v = stack.pop();
        infg.restore(v);
        Color c = SelectColor(v, infg);
        if (c is not defined) {
            actual_spills.append(v);
        }
    }

    return actual_spills;
}
```

Procedura `Select` zdejmuje wierzchołki ze stosu utworzonego przez poprzednią fazę. Przywraca usunięty wierzchołek w grafie interferencji, a następnie wybiera kolor nie kolidujący z sąsiadami. Jeżeli nie da się dobrać koloru to dodajemy wierzchołek do listy rejestrów przeznaczonych do spillingu.

W naszym rozwiązaniu nie przywracamy wierzchołków w grafie interferencji w sposób dosłownych, a zamiast tego na stosie trzymamy wierzchołek oraz listę jego sąsiadów, jakie miał, gdy procedura `Simplify` go kasowała. W celu uproszczenia implementacji nasza procedura `SelectColor` zawsze przydziela jakiś kolor wierzchołkowi. Jeżeli nie da się przydzielić prawidłowego koloru (reprezentowanego przez liczbę mniejszą niż liczba dostępnych rejestrów) to zwraca jakiś nieprawidłowy (reprezentowany przez liczbę większą niż liczba dostępnych rejestrów).

Wstępnie pokolorowane wierzchołki W naszym rozwiązaniu przyjęliśmy, że wierzchołki reprezentujące inne rejestry niż tymczasowe są wstępnie pokolorowane. W dostarczonym szablonie rozwiązania znajdą Państwo kod potrzebny do tego podejścia.

Debugowanie Przy pracowaniu nad tym zadaniem wygodnie będzie uruchamiać kompilator z opcją `--extra-debug`. Ta opcja spowoduje wrzucenie więcej komunikatów do katalogu `xilog`. Dodatkowo pojawią się tam zrzuty grafów w formacie graphviz-a, w tym grafu interferencji, w plikach z końcówką `.xdot`. Mogą Państwo interaktywnie przeglądać te grafy za pomocą programu `xdot`. W plikach z rozszerzeniem `.regmapping` znajdą Państwo wynik alokacji rejestrów. W pliku o nazwie `regalloc.final.ir` znajdą Państwo kod programu z przydzielonymi rejestrami.
Niestety używana przez nas biblioteka `ocamlgraph` potrafi działać wolno, gdy skompilowaliśmy program do bajtkodu. Podczas rozwiązywania zadania proszę zwracać uwagę nie na czas, a na liczbę iteracji głównej pętli. Z komunikatów generowanych przez naszą wtyczkę odczytają Państwo bez trudu liczbę iteracji jaką zrobił nasz algorytm. Jeżeli nasz potrzebuje trzech iteracji, aby przydzielić rejestry, a Państwa rozwiązanie potrzebuje dziesięciu, to jest to niepokojący sygnał. Jeżeli natomiast Państwa rozwiązanie potrzebuje trzech iteracji, a nasze dziesięciu, to prowadzący pracownię będzie wdzięczny za e-mail z takim programem.
Pomocne może być też użycie eksperymentalnej opcji kompilatora `--registers-description=simple_caller`, która ogranicza liczbę dostępnych rejestrów.

Dla ciekawskich Heurystyka kosztów, którą się posługujemy zlicza liczbę transferów z/do pamięci dla każdego rejestru, jaką należałoby wstawić do kodu programu gdyby dany rejestr był wybrany do spillingu. Ma to odzwierciedlić dodatkowy koszt czasowy wykonania programu, gdzie za koszt transferu przyjmujemy jeden. Aby taka analiza kosztów była użyteczna musi uwzględnić również to, że niektóre transfery mogą być wstawione w ciałach pętli, co oznacza że mogą się wykonać wiele razy podczas działania programu. Heurystyka uwzględnia to mnożąc koszt każdego transferu przez 10 do potęgi będącej poziomem zagnieżdżenia instrukcji w pętlach. By poziom zagnieżdżeń obliczyć posługujemy się prostą statyczną analizą *natural loops* wykrywającą pętle w kodzie pośrednim. Taka analiza opiera się o klasyczną analizę przepływu danych – analizę dominacji.
Nasza implementacja spillingu jest bardzo prosta i bywa czasem określana w literaturze jako *Spill Everywhere* ponieważ wstawia transfer z/do pamięci programu przed każdym użyciem tymczasowego rejestru oraz po każdej jego modyfikacji. Skutkiem takiego podejścia mogą być nadmiarowe odczyty pamięci, które dałoby się zlikwidować wykrywając miejsca użycia rejestru, które są blisko siebie. W takich sytuacjach czasami da się przechować wynik transferu w jakimś sprzętowym rejestrze i wyeliminować potrzebę transferu w pobliskim użyciu. Bardziej zaawansowane kompilatory posługują się tutaj dodatkową lokalną analizą, to jest taką co działa w obrębie jednego bloku podstawowego, do wykrywania takich sytuacji. Przykład takiej analizy, użytej do lepszej heurystyki kosztów, znajdziecie Państwo w pracy doktorskiej [1] w rozdziale 8.7.
Proces złączania rejestrów jest bardzo ważnym elementem całej optymalizacji. Dzięki niemu kompilator jest w stanie wykonać tak zwany register targetting. Przykładowo, konwencja wywołań jaką przyjęliśmy wymaga aby wynik funkcji znajdował się w rejestrze $v0, stąd najlepiej byłoby aby ostatnie obliczenia jakie wykonuje funkcja odbywały się od razu w tym rejestrze. Ten efekt jest uzyskiwany właśnie dzięki fazie złączania rejestrów. Więcej o znaczeniu tej fazy znajdą Państwo w książce Muchnicka [2] w rozdziale 16.3.6. Zaawansowany algorytm złączania rejestrów (wpływający na schemat całego algorytmu), zwany iterated coalescencing, znajdziecie państwo w książce Appela [3] w rozdziale 11.
Z mechanizmów obecnych w zaawansowanych wariantach algorytmu alokacji warto wymienić jeszcze rematerializację oraz live range splitting. Pierwszy mechanizm pozwala kompilatorowi wykryć, które obliczenia jest lepiej wykonać od nowa niż odczytywać z pamięci. Drugi mechanizm pozwala podzielić czas życia

tymczasowego rejestru na mniejsze fragmenty, dzięki czemu spilling może odbyć się tylko we fragmencie kodu procedury. Oba mechanizmy są omówione we wspomnianej pracy doktorskiej [1]. W artykule [5] znajdą Państwo informacje o implementacji w kompilatorze gcc.
Alokacja rejestrów oparta o kolorowanie grafu jest bardzo popularnym algorytmem i jej nowoczesne wersje do dzisiaj są używane w wiodących kompilatorach, takich jak gcc. W środowiskach gdzie czas kompilacji odgrywa kluczową rolę stosuje się czasem inne algorytmy. Takimi środowiskami są często kompilatory Just-In Time, to jest takie co kompilują kod dopiero w momencie uruchomienia programu. Są one stosowane w takich językach jak Java czy C#. Czas działania programu uwzględnia w sobie czas kompilacji, którą wykonuje interpreter. Stąd szuka się kompromisu pomiędzy czasem działania algorytmów, a jakością otrzymanego wyniku. Popularnym algorytmem jest tutaj Linear Scan[6]. Wykorzystuje go również kompilator OCamla ocamlopt. Kompilator llvm, będący sercem kompilatora clang, korzysta z jeszcze innego algorytmu alokacji. Ich algorytm zwany układaniem puzli [4] działa tylko z programem w postaci SSA.

Literatura

[1] P. Briggs. Register allocation via graph coloring. Phd 1992.

[2] S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA ©1997.

[3] A. Appel. Modern Compiler Implementation in ML. Cambridge University Press New York, NY, USA ©1997.

[4] F. M. Q. Pereira. J. Palsberg. Register allocation by puzzle solving. PLDI 2008

[5] M. Punjani. Register Rematerialization In GCC. GCC Developers' Summit 2004.

[6] M. Poletto. V. Sarkar. Linear Scan Register Allocation. TOPLAS 1999.