Pamięć notatnikowa, systemy czasu rzeczywistego,
linie krytyczne, szeregowanie zadań

Jerzy R. NAWROCKI*, Wojciech COMPLAK*

ROZMIESZCZANIE ZADAŃ CZASU RZECZYWISTEGO W PAMIĘCI NOTATNIKOWEJ†

Pamięć notatnikowa może znacznie skrócić czas wykonania programu. Niektórzy uważają jej działanie za nieprzewidywalne i zalecają rezygnację z wykorzystania jej w systemach silnie uwarunkowanych czasowo. Podstawowym problemem jest uwzględnienie wpływu przełączania zadań oraz obsługi przerwań na czas wykonania programu. Jednak zaniedbanie przyspieszenia wynikającego z jej działania zmniejsza prawdopodobieństwo istnienia uszeregowania dopuszczalnego, a co za tym idzie uniemożliwia skonstruowanie wielu systemów sterowania. W artykule zaproponowano deterministyczne podejście do wykorzystania pamięci notatnikowej, w którym zamiast statycznej (biernej) analizy programu zastosowano odpowiednie metody umożliwiające kontrolowanie jej zawartości. Przedstawiono dwie metody ładowania kodu do pamięci notatnikowej z oddzielną pamięcią danych i kodu: metodę nanizania bloków podstawowych oraz metodę nanizania linii pamięci notatnikowej. Zwykle rozmiar pamięci notatnikowej jest istotnie mniejszy od rozmiaru programu, a zatem istnieje problem określenia tych obszarów kodu, które mają być załadowane do pamięci notatnikowej (tzw. rezydentów pamięci notatnikowej). Ponieważ problem ten jest trudny obliczeniowo, zaproponowano odpowiedni algorytm heurystyczny.

1 WSTĘP

W systemach czasu rzeczywistego nie wystarcza, aby program dostarczał poprawne rezultaty. Musi je również dostarczyć na czas. Współczesne komputerowe systemy silnie uwarunkowane czasowo składają się z kilkuset, a nawet kilku tysięcy zadań. Zagwarantowanie, że każde zadanie zawsze zdąży wytworzyć odpowiedź przed swoimi liniami krytycznymi, wymaga oszacowania maksymalnego czasu wykonania każdego zadania, a następnie zbudowania statycznego uszeregowania wszystkich zadań.

Współczesne komputery wyposażane są w pamięć notatnikową, której zadaniem

* Instytut Informatyki, Politechnika Poznańska, ul. Piotrowo 3A, 60-965 Poznań, e-mail: Jerzy.Nawrocki@put.poznan.pl, Wojciech.Complak@cs.put.poznan.pl

† Praca wykonana w ramach grantu 91-378/BW

jest zwiększenie wydajności systemu poprzez pośredniczenie między szybkimi procesorami a stosunkowo wolną pamięcią operacyjną. Niektórzy autorzy uważają jednak zachowanie pamięci notatnikowej za tak dalece nieprzewidywalne, że sugerują rezygnację z jej wykorzystania w systemach czasu rzeczywistego [7, 9]. Nie jest to jednak dobry pomysł. Z przeprowadzonych eksperymentów wynika, że użycie pamięci notatnikowej może zredukować czas wykonania zadania nawet 50-krotnie [2].

Podstawowy, z punktu widzenia systemów czasu rzeczywistego, problem związany z pamięcią notatnikową polega na uwzględnieniu wpływu przełączania zadań na maksymalny czas wykonania. Przełączanie zadań może prowadzić do zmiany zawartości pamięci notatnikowej. Nowouruchomione zadanie może wymagać ściągnięcia do pamięci notatnikowej nowych danych i instrukcji, co wiąże się z pewnym opóźnieniem wykonania instrukcji. W tej sytuacji szacowanie maksymalnego czasu wykonania jest obarczone istotnym błędem (oszacowanie jest zwykle dużo większe niż faktyczny czas wykonania). Problem staje się jeszcze trudniejszy, gdy należy uwzględnić przerwania. Mają one charakter niedeterministyczny. Mogą się pojawić w dowolnym momencie i zainicjować wykonanie bardzo różnych fragmentów oprogramowania. W rezultacie wymagałoby to przyjęcia, że każda dana i instrukcja musi być ściągnięta do pamięci notatnikowej przed jej wykonaniem. Oczywiście, takie oszacowanie byłoby z praktycznego punktu widzenia bez sensu. Ale przerwań nie można zaniedbać (np. blokując je). Pozostaje zatem problem ujarzmienia niedeterminizmu, którego głównym źródłem jest właśnie system przerwań. Należy znaleźć rozwiązanie, które pozwoliłoby korzystać z pamięci notatnikowej bez konieczności tolerowania nierealnych oszacowań maksymalnego czasu wykonania.

W artykule zaproponowano deterministyczne podejście do zarządzania pamięcią notatnikową w systemach silnie uwarunkowanych czasowo, które w połączeniu ze statycznym modelem szeregowania zadań, pozwala na uwzględnienie mechanizmu przerwań i uniknięcie problemu nierealnych oszacowań maksymalnego czasu wykonania programów. Przed uruchomieniem systemu określany jest tzw. zbiór rezydentów pamięci notatnikowej, czyli tych podprogramów i danych, które przez cały czas będą znajdowały się w pamięci notatnikowej i nie będą z niej wypierane. W pracy przedstawiono algorytm wyboru rezydentów pamięci notatnikowej oraz zaproponowano metody ładowania podprogramów i danych do pamięci notatnikowej dla różnych architektur komputerów.

2 ŁADOWANIE KODU I DANYCH DO PAMIĘCI NOTATNIKOWEJ

Istnieją dwa podejścia do problemu efektywnego zarządzania zawartością pamięci notatnikowej – pasywne i aktywne.

W metodzie pasywnej, zazwyczaj stosowanej we współczesnych architekturach, zakłada się, że najważniejszym wymaganiem stawianym pamięci notatnikowej jest zapewnienie pełnej przezroczystości jej działania z punktu widzenia wykonywanych programów. W podejściu tym zarządzanie zawartością pamięci notatnikowej realizowane jest wyłącznie przez mechanizmy sprzętowe odpowiedniego kontrolera.

Metoda aktywna jest rozszerzeniem metody pasywnej o dodatkowe instrukcje ste-

rujące kontrolerem pamięci notatnikowej (np. PowerPC [12]). Podstawowym celem wprowadzenia dodatkowych mechanizmów programowych było uzyskanie wyższej, niż osiągalna w metodzie pasywnej, wydajności. Dzięki takim instrukcjom można arbitralnie załadować i utrzymywać w pamięci notatnikowej często wykorzystywane bloki pamięci, a mniej użyteczne usuwać. Są one zazwyczaj generowane przez kompilator. Jeśli jednak program nie zawiera takich rozkazów, to kontrolowanie zawartości pamięci notatnikowej realizowane jest tak jak w metodzie pasywnej. Możliwość sterowania zawartością pamięci notatnikowej jest dużym ułatwieniem dla konstruktorów systemów silnie uwarunkowanych czasowo, w których chcemy mieć możliwość decydowania o tym, czy i kiedy dany blok zostanie załadowany albo wyładowany z pamięci notatnikowej.

2.1 STATYCZNA ANALIZA PROGRAMÓW

Porównanie istniejących metod szacowania maksymalnego czasu wykonania programów [2, 3] wskazuje jednoznacznie, że jedyną wiarygodną, z punktu widzenia systemów silnie uwarunkowanych czasowo, grupą metod są metody analityczne, czyli takie, w których szacowanie wykonywane jest poprzez przeglądanie tekstu programu na poziomie języka źródłowego, docelowego lub na obu poziomach. Podstawową trudnością w uzyskaniu dokładnych wyników oszacowania maksymalnego czasu wykonania stanowi uwzględnienie zjawiska wypierania zawartości pamięci notatnikowej. Zjawisko to, wynikające przede wszystkim z ograniczonej pojemności pamięci notatnikowej (i w mniejszym stopniu ze skończonej asocjacyjności), występuje w dwu postaciach ([6]):

- wypierania międzyzadaniowego (ang. *intertask*) – w praktyce przyjmuje się, że każde przełączenie pomiędzy zadaniami wymusza przeładowanie całej zawartość pamięci notatnikowej, co prowadzi do uzyskania pesymistycznego oszacowania. Proponowane rozwiązania wymagają albo niestandardowego wsparcia sprzętowego [8] albo stosują trudne w implementacji rozwiązania programowe [14];
- wypierania wewnątrzzadaniowego (ang. *intratask*) – przy założeniu, że mamy do czynienia tylko z zadaniami niepodzielnymi, statyczna analiza daje dobre oszacowania maksymalnych kosztów czasowych. Założenie takie jest jednak nierealistyczne, gdyż rzeczywisty system musi odbierać asynchroniczne sygnały od zewnętrznych urządzeń, przekazywane za pomocą przerwań. Uwzględnienie spowodowanego w ten sposób spowolnienia przetwarzania prowadzi do uzyskania bardzo pesymistycznych rezultatów [5]. Może się bowiem zdarzyć, iż intensywna obliczeniowo pętla jest wielokrotnie przerywana nadchodzącymi przerwaniami i należy założyć, że za każdym razem tracona jest cała zawartość pamięci notatnikowej.

2.2 PROGRAMOWE ZARZĄDZANIE ZAWARTOŚCIĄ PAMIĘCI NOTATNIKOWEJ

W większości architektur komputerów wyposażonych w pamięć notatnikową, kod i dane ładowane są do niej w chwili, gdy następuje do nich odwołanie. Jeśli chcemy sami decydować o jej zawartości musimy rozwiązać dwa problemy:

- jak zabronić ładowania niepożądanych elementów do pamięci notatnikowej, a co za

tym idzie wypierania tych fragmentów pamięci, które mają w niej pozostać ?

- jak załadować wybrane dane i kod do pamięci notatnikowej ?

Rozwiązanie pierwszego problemu jest stosunkowo łatwe. W popularnych procesorach wyposażonych w pamięć notatnikową (Pentium [11], PowerPC [12]) możliwe jest zakazanie ładowania określonych obszarów do pamięci notatnikowej.

Rozwiązanie drugiego problemu jest bardziej kłopotliwe. W istocie chodzi tutaj o wyeliminowanie kar czasowych za nietrafienie w pamięć notatnikową. W literaturze znaleźć można, co prawda, cały szereg podejść (szerzej omówione w [2]) proponujących różne rozwiązania tak programowe, jak i wykorzystujące dodatkowe wsparcie sprzętowe. Metody te jednak są mało użyteczne w systemach silnie uwarunkowanych czasowo, ponieważ są zorientowane na redukcję współczynnika nietrafień (który ma charakter statystyczny) i domyślne zastosowanie w dynamicznym modelu szeregowania zadań.

Istniejące ograniczenia można jednak obejść stosując odpowiedni – statyczny model szeregowania zadań i aktywnie zarządzając zawartością pamięci notatnikowej. Sterowanie takie jest stosunkowo łatwe, jeżeli dysponujemy odpowiednio zaawansowanym procesorem i skorzystamy z rozkazów systemowych zarządzających pamięcią notatnikową. W przypadku procesorów nie wyposażonych w takie mechanizmy konieczne jest użycie innych technik, aby umieścić wybrany kod i dane w pamięci notatnikowej.

Załadowanie danych do pamięci notatnikowej nie stanowi problemu, ponieważ można tego dokonać poprzez ich odczytanie. Podobnie można postąpić z kodem, jeśli mamy do czynienia ze zunifikowaną pamięć notatnikową. Załadowanie kodu jest jednak zdecydowanie trudniejsze w przypadku nowszych procesorów, których pamięci notatnikowe są rozdzielne, a co za tym idzie nie można odczytać kodu bez jego wykonywania. W niektórych, przejściowych konstrukcjach (na przykład w Pentium [11]) istnieje możliwość obejścia tego ograniczenia, dla zachowania zgodności ze starszym oprogramowaniem, jednak oficjalnie nie przewiduje się implementowania takiego mechanizmu w następnych generacjach procesorów. Nieuniknione jest więc zmodyfikowanie kodu tak, aby dodać dodatkową ścieżkę przepływu sterowania, która wymusi załadowanie go do pamięci notatnikowej. Modyfikacja taka musi jednak zachować semantykę oryginalnego programu, który może zawierać instrukcje warunkowe i instrukcje z efektami ubocznymi (np. wysłanie danych do urządzenia zewnętrznego).

W zależności od organizacji pamięci notatnikowej zaproponować można trzy metody ładowania kodu:

- jeżeli pamięć jest zunifikowana, to zarówno dla kodu, jak i dla danych można wykorzystać proste ściąganie zawartości,
- jeżeli pamięć jest rozdzielna można odpowiednio zmodyfikować oryginalny kod programu wykorzystując nanizanie linii pamięci notatnikowej, w celu załadowania kodu do pamięci notatnikowej,
- jeżeli pamięć jest rozdzielna i nie wiemy, na którym procesorze danej rodziny (czyli jaka będzie wielkość linii pamięci notatnikowej) dany program będzie wykonywany, można wykorzystać najbardziej ogólną metodę – nanizanie bloków podstawowych.

2.3 PROSTE ŚCIĄGANIE ZAWARTOŚCI

Koncepcja prostego ściągania zawartości opiera się na spostrzeżeniu, że aby załadować dane do pamięci notatnikowej wystarczy je odczytać. Takie podejście można zastosować w przypadku zunifikowanej pamięci notatnikowej zarówno do danych, jak i do kodu. Korzystamy tutaj z faktu, iż w koncepcji von Neumanna kod i dane nie są rozróżniane, a zatem pamięć notatnikowa traktuje tak samo wykonanie, jak i odczyt kodu.

Załóżmy, że obszar pamięci, który ma zostać załadowany do pamięci notatnikowej zaczyna się od adresu `start`, ma rozmiar `bytes_no` wyrażony w bajtach i będzie wykonywany na procesorze o rozmiarze linii pamięci notatnikowej `line` (również w bajtach). Analizując mechanizm działania pamięci notatnikowej można dojść do wniosku, że wystarczy odczytać jedną daną (bajt) z każdej linii pamięci notatnikowej zajmowanej przez kod, który ma zostać załadowany. Oznacza to, że należy wykonać dokładnie:

$$n = \frac{start + bytes\_no - 1}{line} - \frac{start}{line} + 1$$

iteracji. Algorytm wykorzystujący takie podejście przedstawiono na rys. 1.

```
first_line=start/line;  last_line=(start+bytes_no-1)/line;
for(i=first_line,j=start;i<=last_line;i++,j+=line)
    mem_read_byte(j);
```

Rys. 1. Algorytm ładowania linii pamięci notatnikowej

Fig 1. Loading cache lines

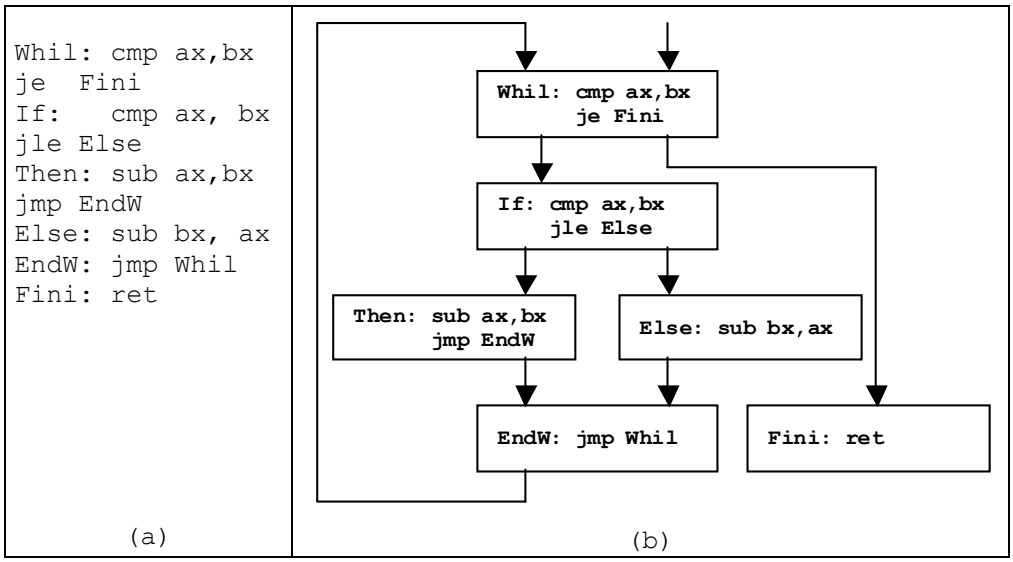
2.4 METODA NANIZANIA BLOKÓW PODSTAWOWYCH

Zastosowanie metody prostego ściągania zawartości jest niemożliwe, gdy chcemy załadować fragment kodu, a procesor docelowy korzysta z rozdzielnej pamięci notatnikowej kodu i danych i nie posiada instrukcji wstępnego ładowania. Jedynym rozwiązaniem jest wówczas załadowanie kodu poprzez jego wykonanie. Nie możemy jednak po prostu wykonać segmentu kodu w niezmienionej postaci z dwu powodów:

- kod często powoduje powstanie efektów ubocznych (stan urządzeń, dane pośrednie), których nie możemy anulować po zakończeniu ładowania, a przed rzeczywistym uruchomieniem zadania,
- aby zagwarantować maksymalny czas wykonania oszacowany przy założeniu, że kod zostanie załadowany do pamięci notatnikowej musimy zagwarantować przejście (wykonanie) na etapie ładowania przez wszystkie bloki kodu – powstaje więc pytanie jak zagwarantować, że przy ładowaniu przejdziemy wszystkie możliwe ścieżki przepływu sterowania (np. obydwie ścieżki instrukcji warunkowej `if-then-else`).

Aby oba te problemy rozwiązać musimy tak zmodyfikować kod, aby móc wymusić przepływ sterowania przez wszystkie bloki w momencie ładowania i nie wykonywać rozkazów, które mogłyby zmienić stan systemu. Musimy więc mieć możliwość odróżnienia za pomocą dodatkowych informacji przebiegu ładującego od przebiegu użytecznego. Odpowiednie modyfikacje muszą być wprowadzone na poziomie tak zwanych bloków

podstawowych. W uproszczeniu można powiedzieć, że blok podstawowy jest to fragment kodu (w języku asemblera lub w języku pośrednim) bez instrukcji skoku w środku (szersze omówienie znaleźć można w [1]). Z bloków podstawowych wyodrębnionych w analizowanym fragmencie kodu tworzy się graf przepływu. Graf przepływu jest grafem skierowanym, w którym wierzchołki odpowiadają blokom podstawowym, a łuki przepływom sterowania pomiędzy nimi. Przykładowy program w języku asemblera i odpowiadający mu graf przepływu pokazano na rys. 2.



Rys. 2. Przykładowy program (a) i odpowiadający mu graf przepływu (b).

Fig 2. An example of a program (a) and corresponding flow graph (b).

Problem polega na wymuszeniu dokładnie jednokrotnego przejścia przez każdy blok podstawowy. Jeżeli z wierzchołka A wychodzi tylko jeden łuk, to można wykorzystać to naturalne przejście, chyba że jest to łuk zwrotny (terminem tym określa się w teorii kompilacji łuk „realizujący" pętlę – na rys. 2b łuk `EndW→Whil` jest łukiem zwrotnym). Jeśli z wierzchołka A wychodzą 2 łuki, to należałoby wymusić przejście jednym z nich, zostawiając przejście drugim na później. Końcowy blok podstawowy – ze względu na zawartą w nim instrukcję powrotu `RET` – powinien być wykonywany jako ostatni. Na przykład dla grafu przepływu z rys. 2b bloki podstawowe mogłyby być wykonane w następującej kolejności `Whil` ⇒ `If` ⇒ `Then` → `EndW` ⇒ `Else` ⇒ `Fini`.

Na rys. 3 przedstawiono program z rys. 2 zmodyfikowany według przedstawionej koncepcji (zmienna `LOAD` pozwala na rozróżnienie faz ładowania i wykonywania). Modyfikacje zaznaczono wytłuszczoną czcionką. Jak widać, liczba dodatkowych instrukcji jest bardzo duża i w istotny sposób zwiększa rozmiar programu oraz jego czas wykonania. Ze względu na koszty metodę tę należałoby stosować w sytuacjach, gdy mamy do czynienia z relatywnie dużymi blokami podstawowymi, a liczba wymuszanych przejść jest relatywnie mała. Innym problemem są trudne instrukcje, takie jak CALL, dzielenie (ponieważ w fazie ładowania bloki wykonywane są w nienaturalnym porządku, może się

zdarzyć, że dzielnik będzie miał wartość zero) lub też instrukcje wejścia/wyjścia.

Dodatkowym ograniczeniem tej metody jest fakt, iż nawet wymuszenie wykonania wszystkich bloków podstawowych nie zapewnia załadowania ich w całości do pamięci notatnikowej. Duże bloki podstawowe mogą zajmować obszar pamięci odwzorowywany w więcej niż jedną linię pamięci notatnikowej, a co za tym idzie należałoby dokonać dodatkowych modyfikacji gwarantujących załadowanie wszystkich linii odpowiadających danemu blokowi podstawowemu.

```
      JMP Whil
Whi1: CMP LOAD,TRUE
      JE  Else
Whil: CMP AX,BX
      PUSHF
      CMP LOAD,TRUE
      JNE Els1
      SUB SP,2
      CMP JEDEN,0
      JMP Skok
```

```
Els1: POPF
      je Fini
If:   cmp ax, bx
      PUSHF
      CMP LOAD,TRUE
      JNE Els1
      SUB SP,2
      CMP JEDEN, 0
      JMP Skok
```

```
Els1: POPF
      jle Else
Then: sub ax,bx
      jmp EndW
Else: sub bx,ax
      CMP LOAD,TRUE
      JE Fini
EndW: jmp Whi1
Fini: ret
```

Rys. 3. Zmodyfikowany program z rys. 2

Fig 3. Modified program from fig. 2

2.5 METODA NANIZANIA BLOKÓW PAMIĘCI

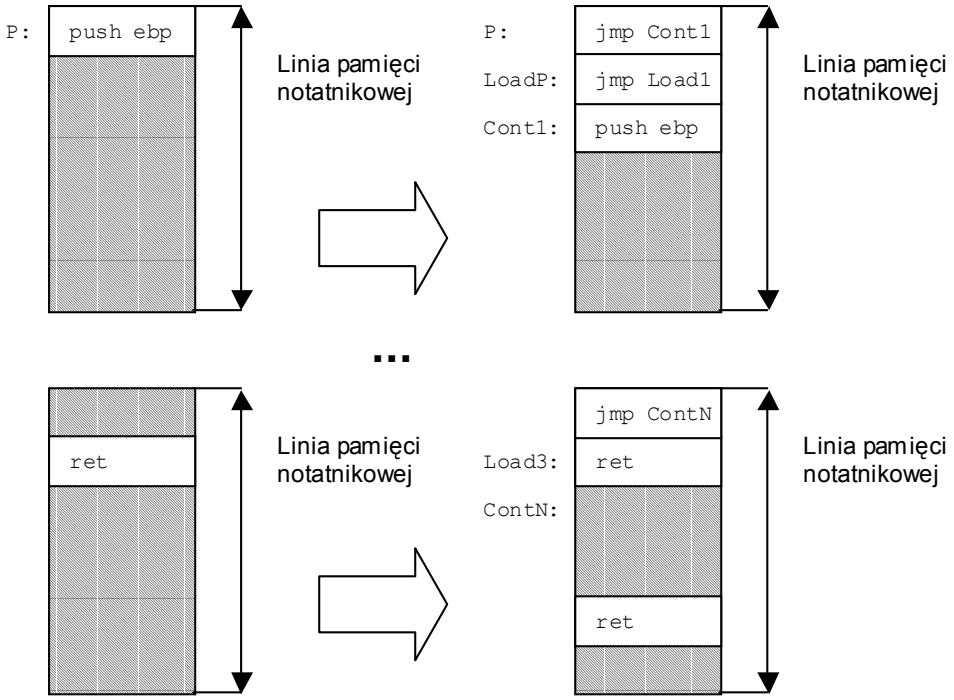
Efektywniejsze podejście do problemu ładowania kodu opiera się na spostrzeżeniu, że kod (i dane) są zawsze ładowane do pamięci notatnikowej tzw. liniami. Linia ma dla danego procesora stały rozmiar będący zawsze naturalną potęgą 2 (np. w *Pentium* jej rozmiar wynosi 32 bajty [11]).

Jak można zauważyć (rys. 4), metoda ta jest bardzo prosta i przy założeniu (w praktyce zawsze spełnionym), że rozmiar linii pamięci notatnikowej jest odpowiednio duży, związane z tą metodą narzuty pamięciowe i czasowe są niewielkie. Jednak, gdyby linia była bardzo mała w porównaniu z kodem dwóch rozkazów skoku bezwarunkowego, to metoda ta byłaby nieefektywna. W skrajnym przypadku (gdyby linia mieściła nie więcej niż wymagane dwie instrukcje skoku) stosowanie jej byłoby w ogóle niemożliwe i wówczas należałoby zastosować metodę nanizania bloków podstawowych.

3 DWUFAZOWE PODEJŚCIE DO SZEREGOWANIA ZADAŃ

Jedną z metod statycznego szeregowania jest minimalizacja cyklicznego obciążania przedstawiona po raz pierwszy w pracy [13]. W podejściu tym szereguje się zbiór niewywłaszczalnych cyklicznych zadań (metodykę przekształcania innych typów zadań do zadań cyklicznych znaleźć można w [4]) na jednym procesorze. Czas podzielony jest na ciąg ram czasowych o takim samym rozmiarze oznaczanym symbolem τ, które są

numerowane kolejnymi nieujemnymi liczbami całkowitymi 0, 1, 2, … Okres każdego zadania jest wielokrotnością τ. Jeśli okres zadania T_i wynosi $\pi\tau$ (π = 1, 2, 3, …), to współczynnik π_i nazywany jest okresem względnym zadania *T*. Zadanie *T* jest uruchamiane (czyli tworzone są kolejne instancje) w ramach $f (0 \le f \le \pi), f + \pi, f + 2\pi, f + 3\pi$ itd. *f* nazywane jest fazą zadania *T*. Wartość π zależy od prędkości działania procesu, który ma być sterowany bądź monitorowany przez zadanie *T*. Przez d_i oznaczamy czas trwania zadania T_i (jest to maksymalny czas wykonania liczony po wszystkich instancjach zadania T_i). Obciążenie ramy czasowej *F*[*j*], oznaczane jako *y*[*j*], równe jest sumie czasów trwania wszystkich instancji zadań przypisanych do ramy *F*[*j*]. Maksymalne obciążenie ramy *z* równe jest $max\{F[0], F[1], \ldots, F[H-1]\}$.



Rys. 4. Schemat wplatania łańcucha skoków bezwarunkowych w kod podprogramu (zaciemnione obszary oznaczają oryginalne instrukcje podprogramu).

Fig 4. Cache lines threading framework (dark areas correspond to original program code)

Problem minimalizacji cyklicznego obciążania polega na określeniu fazy f_i każdego zadania T_i tak, aby zminimalizować maksymalne obciążenie ram. Przyjmuje się, że niedopuszczalne jest przekraczanie przez jakiekolwiek zadanie granic ram czasowych (zadania dłuższe niż τ, dzielone są na ciąg zadań szeregowany z uwzględnieniem ograniczeń kolejnościowych). Problem minimalizacji cyklicznego obciążania jest silnie NP-trudny [10]. Jednak dzięki takiemu modelowi szeregowania zadań, można skonstruować efektywne podejście do zarządzania zawartością pamięci notatnikowej, które radzi sobie z problemem przerwań. Wystarczy, jeśli przed uruchomieniem systemu określimy

zbiór zadań, które mają być załadowane do pamięci notatnikowej (tzw. rezydentów pamięci notatnikowej), a następnie w prologu uruchamiania systemu załadujemy je i zablokujemy w pamięci notatnikowej. Pozostaje jedynie do rozwiązania problem określenia zbioru rezydentów pamięci notatnikowej. Można go rozwiązać opierając się na koncepcji ulepszania statycznego uszeregowania.

Zaproponowane podejście jest dwufazowe. W pierwszej fazie konstruowane jest uszeregowanie początkowe zgodnie z optymalizacyjną wersją problemu cyklicznego obciążania. Czas trwania d_i każdego zadania T_i szacowany jest przy założeniu, że zadanie w trakcie wykonywania nie znajdzie się w pamięci notatnikowej. W wyniku wykonania tej fazy otrzymuje się sekwencję ram czasowych z przypisanymi im instancjami zadań.

W drugiej fazie rozważamy możliwość załadowania poszczególnych fragmentów systemu do pamięci notatnikowej. Z każdym elementem e_j związany jest jego rozmiar s_j odpowiadający liczbie komórek, jaką zajmie w pamięci notatnikowej. C to rozmiar pamięci notatnikowej, a sumaryczna liczba komórek jakie zajmą wszystkie załadowane elementy nie może przekroczyć C. Jak można wykazać ([2]) problem ten należy do klasy NP i można skonstruować wielomianową transformację z problemu plecakowego.

Dla klasycznego problemu plecakowego istnieje prosta heurystyka, która najpierw porządkuje elementy według nie rosnących wartości proporcji v_i/w_i, a następnie zgodnie z tym porządkiem ładuje elementy. W opisywanym problemie każdy element i opisany jest wektorem wartości v_{ii}, …, v_{mi}. i należy zwrócić uwagę, że z punktu widzenia j-tego kryterium, wartość v_{ji} aktualnie rozważanego elementu i może być większa niż jest to wymagane. Dlatego elementy należy sortować według następującego kryterium:

$$\sum_{j=1,\ldots,m} \frac{\min\{v_{ji}, Xv_j\}}{w_i}$$

gdzie Xv_j jest sumą wartości elementów, które muszą być umieszczone w plecaku, aby spełnić j-te ograniczenie.

Eksperymentalna ocena jakości heurystyki, wykonana na przykładzie jądra systemu MINIX ([2]), dowiodła jej użyteczności w praktycznych przypadkach, w których zastosowanie algorytmu dokładnego jest ze względu na koszty czasowe nierealne.

4 PODSUMOWANIE

W artykule zaprezentowano deterministyczne podejście do zarządzania pamięcią notatnikową, które pozwala na jej wykorzystywanie w systemach silnie uwarunkowanych czasowo. Zaprezentowany sposób obsługi przerwań, przy przyjęciu statycznego modelu szeregowania, gwarantuje całkowitą przewidywalność wpływu działania pamięci notatnikowej na prędkość przetwarzania oraz możliwość sprawdzenia przed uruchomieniem systemu czy każde zadanie zawsze zdąży wytworzyć odpowiedź przed swoimi liniami krytycznymi. W pracy przedstawiono problem wyboru rezydentów pamięci notatnikowej, wykazano jego NP-zupełność i zaproponowano algorytm heurystyczny.

[1] AHO A., et al., Compilers, Principles, Techniques and Tools, Reading, MA: Addison Wesley, 1988.
[2] COMPLAK W., Deterministyczne podejście do zarządzania pamięcią notatnikową w systemach silnie uwarunkowanych czasowo, Rozprawa doktorska, Politechnika Poznańska, 2001.
[3] COMPLAK W., Szacowanie czasu wykonania programów, I Krajowa Konferencja Metody i systemy komputerowe w badaniach naukowych i projektowaniu inżynierskim, Kraków 1997.
[4] CZAJKA A., Statyczne szeregowanie zadań o okresach binarnych w systemach silnie uwarunkowanych czasowo, Rozprawa doktorska, Politechnika Poznańska, 2000.
[5] FERDINAND C., MARTIN F., WILHELM R., ALT M., Cache Behavior Prediction by Abstract Interpretation, Report 05/97.
[6] HANDY J., the Cache Memory book, second edition, Academic Press, 1998.
[7] HALANG W. A., STOYENKO A. D., Constructing predictable realtime systems, Kluwer Academic Publishers, 1991.
[8] KIRK D. B., SMART cache design, Proc. 10th IEEE Real-Time Systems Symp., 1989, 229-237.
[9] MOORE S. W., Scalable temporally predictable memory structures, IEEE Real-time Applications Workshop, March, 1994.
[10] NAWROCKI J., CZAJKA A., Szeregowanie zadań w systemach silnie uwarunkowanych czasowo metodą cyklicznego obciążania, Zeszyty Politechniki Śląskiej, No. 1389 (1998), 169-179.
[11] Pentium® Processor Family Developer's Manual, Intel Corp., 1997, www.intel.com.
[12] PowerPC™ 604 RISC Microprocessor Technical Summary, Motorola Inc., 1994, www.mot.com.
[13] SCHWEITZER P. J., DROR M., TRUDEAU P., The periodic loading problem: formulation and heuristics, INFOR, vol. 26 (1988), No.1, 40-62.
[14] WOLFE A., Software-based cache partitioning for real-time applications, Proc. Third Workshop Responsive Computer Systems, Sept. 1993.

PLACEMENT OF REAL-TIME TASKS IN CACHE MEMORY

Cache memory may significantly improve program performance. However it is still believed to be unpredictable and thus in the area of HRT applications considered a nuisance. The main reason for it is the impact of task switching and interrupt handling causing intertask and intratask cache thrashing on worst case execution time. On the other hand neglecting possible speedup accomplished due to cache memory reduces possibility of constructing a feasible solution for many control systems.

In the paper we propose a new deterministic approach to control the contents of cache memory. Two algorithms for loading code into split caches were shown: basic block threading and cache lines threading. Usually cache memory is significantly smaller than the size of program segments and therefore the problem of choosing objects which should be locked in the cache memory (so called residents) to get a feasible schedule arises. We formulate the problem of periodic loading with cache memory and show its NP-completeness. In the end a simple heuristic for the problem is proposed.