INSTRUCTION AND CYCLE ACCURATE DEBUGGER

Tomáš Korec
Master Degree Programme (1), FIT BUT
E-mail: xkorec00@stud.fit.vutbr.cz

Supervised by: Zdeněk Přikryl
E-mail: iprikryl@fit.vutbr.cz

Abstract: This paper deals with instruction-accurate debugging using a cycle-accurate simulation. The cycle-accurate simulation (a lower level of abstraction) is necessary for microarchitecture debugging (e.g. pipeline debugging). In this case, one instruction takes more than one cycle, but for application debugging (a higher level of abstraction) we need one instruction to be the smallest step of the debugger. The paper describes concepts of how to achieve it. One of them was successfully implemented and the goal of making the debugger supporting both levels of abstraction was achieved.

Keywords: debugger, instruction-accurate simulation, cycle-accurate simulation, simulator

1 ÚVOD

Táto práca sa zaoberá implementáciou ladiaceho nástroja [1] na úrovni inštrukcií (instruction-accurate debugger) pre simulátor procesorov na úrovni cyklu (cycle-accurate simulator). Ide o návrh a implementáciu novej funkcionality do už existujúceho riešenia projektu Lissom.

Motiváciou tejto práce je nemožnosť pohodlného ladenia softwaru na simulátoroch na úrovni cyklu stávajúcim riešením. Ladiaci nástroj bol doposiaľ stavaný pre použitie so simulátormi na úrovni inštrukcií. So simulátormi na úrovni cyklu, kde sa inštrukcie môžu vykonávať viacej taktov, je beh simulátora zastavovaný každý takt, kedy je inštrukcia vykonávaná. V prípade procesorov s prúdovým spracovaním inštrukcií (pipeline processing) sa tento problém prejavuje najviac. Následkom toho je existujúci ladiaci nástroj pre tieto procesory takmer nepoužiteľný.

Cieľom tejto práce je teda doimplementovať podporu ladenia softwaru pre simulátory procesorov na úrovni cyklu tak, aby bolo možné počas ladenia pohodlne meniť úroveň ladenia medzi inštrukčnou úrovňou a úrovňou cyklu.

2 SÚČASNÝ STAV

V rámci projektu Lissom bol vyvinutý jazyk ISAC [3]. Ide o zmiešaný jazyk na popis architektúry (Architecture Description Language). Pomocou neho môžeme popísať ako mikroarchitektúru, tak aj inštrukčnú sadu procesorov. Môžeme teda mať modely na dvoch úrovniach abstrakcie. Na základe modelu je vygenerovaný kompletný balík nastrojov pre daný procesor. Vygenerovaný je simulátor, ladiaci nástroj, ktorý je súčasťou simulátora, assembler, disassembler a prekladač jazyka C.

Súčasné riešenie projektu Lissom, simulátor a ladiaci nástroj, umožňuje bez problémov simulovať a ladiť software pre procesory popísané na inštrukčnej úrovni. U procesorov popísaných na úrovni cyklu je možné ladiť mikroarchitektúru (čo je výhoda napríklad oproti *GNU Debuggeru*, ktorý to neumožňuje), ale ak chceme ladiť software, nastávajú komplikácie popísané v úvode práce.

3 MOŽNOSTI RIEŠENIA

Riešenie tohto problému spočíva v zaistení vykonania jednej inštrukcie ako najmenšieho kroku ladiaceho nástroja pri ladení [2]. Toto je problém, pretože u modelov na úrovni cyklu je najmenší krok

simulácie jeden takt a jedna inštrukcia sa môže vykonávať viac taktov. Dôležité je teda počkať, než sa inštrukcie pred inštrukciou s breakpointom (zarážka pre ladiaci nástroj) dokončia. Existuje niekoľko riešení. Všetky spočívajú v zásahu do určitej časti generovaného kódu pre simulátor a ladiaci nástroj.

3.1 Prekladanie prázdnymi inštrukciami

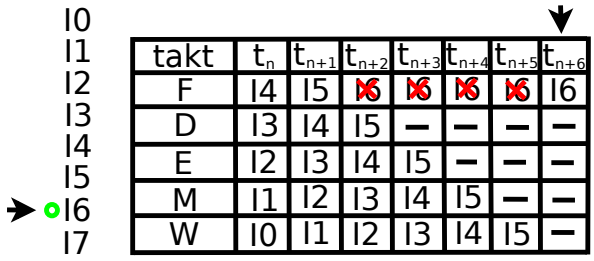
Toto riešenie vychádza z predpokladu, že ak medzi každé dve inštrukcie vložíme dostatočný počet prázdnych inštrukcií (NOP – no operation), bude aj v prípade modelu procesora s prúdovým spracovaním spracovávaná vždy len jedna pôvodná inštrukcia (ostatné budú vložené NOP, ktoré neovplyvnia spracovávanie pôvodnej inštrukcie).

3.2 Zasah do pamäte

Zasahovanie ladiaceho nástroja do pamäte, do inštrukčnej časti, a jej pozmenenie podľa potreby predstavuje ďalšiu možnosť riešenia. Ide o vytvorenie posuvného okna, ktoré by sa do pamäte vkladalo po narazení na breakpoint. Toto okno by obsahovalo kódy prázdnych inštrukcií a vďaka nemu by aj v prípade modelu procesora s prúdovým spracovaním bola spracovávaná len jedna inštrukcia. Okno by sa posúvalo pri krokovaní. Pôvodný obsah pamäte by musel byť uschovaný do pomocnej pamäte tak, aby mohol byť obnovený. Pri tomto riešení je potrebné kontrolovať skokové inštrukcie, aby sa ladený program pri skoku do odloženej časti pamäte choval korektne.

3.3 Čakacie cykly

Ďalšou možnosťou, ako vyriešiť tento problém, je pomocou čakacích cyklov. Po narazení na breakpoint je potrebné uschovať programový čítač (program counter) a v prípade modelu procesora s prúdovým spracovaním vyprázdniť operáciou clear prvý stupeň pipeline, ktorý sa stará o prenos inštrukcie z pamäte do procesora (instruction fetch). Hodnota programového čítača sa neskôr obnoví a program pokračuje ďalej. Čakacích cyklov je potreba niekoľko, aby sa zaistilo, že inštrukcie predchádzajúce breakpoint budú dokončené pred predaním riadenia užívateľovi. Počet čakacích cyklov je odvodený od počtu stupňov pipeline a od maximálneho možného oneskorenia vykonania inštrukcie následkom hazardov. Vďaka tomu sa o hazardy už ďalej starať nemusíme. Riešenie ilustruje obrázok 1. Na obrázku je znázornená pipeline, šípka značí, kde sa nachádza simulácia, I6 je inštrukcia s breakpointom.



Obrázek 1: Vkladanie čakania do pipeline

4 VÝBER RIEŠENIA

Prvé riešenie nie je veľmi náročné na implementáciu, avšak kvôli jeho vplyvu na výkon simulácie bolo zavrhnuté. Vložením *N* inštrukcií by simulácia každej pôvodnej inštrukcie bola *N*-krát pomalšia, bez ohľadu na breakpointy alebo krokovanie. Ostatné riešenia tento problém nemajú.

Druhé riešenie sa pre súčasné riešenie projektu Lissom ukázalo ako nerealizovateľné. Dôvodom je neschopnosť pri zásahoch do pamäte rozoznať, kedy vrátiť prázdnu inštrukciu (inštrukcia pred breakpointom ešte nie je dokončená) a kedy posunúť okno a vrátiť platný obsah pamäte (predchádzajúca inštrukcia je už spracovaná). Pre aplikáciu tohoto riešenia by bolo potrebné upraviť veľkú časť už zavedeného riešenia. Táto úprava by bola veľmi náročná, preto toto riešenie neprichádzalo do úvahy.

Posledné riešenie sa dá považovať za funkčné a je nasaditelné pre modely procesorov nepodporujúcich skokové inštrukcie. Riešenie má totiž s nimi problém. Schopnosť detekcie skokových inštrukcií je pre ladiaci nástroj nevyhnutná, aby bolo možné určiť, ktorou inštrukciou pokračovať. Bez tejto schopnosti by bola vždy obnovená uschovaná hodnota programového čítača a bola by vykonaná bezprostredne nasledujúca inštrukcia. Program by takto počas ladenia nadobudol neplatný stav, pretože by ignoroval výsledok skokovej inštrukcie (zmenu programového čítača).

Predchádzajúce riešenie bolo potrebné upraviť tak, aby ladiaci nástroj dokázal zistiť, že došlo k skoku. To bolo dosiahnuté odlišným zaobchádzaním s programovým čítačom. Jeho hodnota sa po narazení na breakpoint nastaví na určitú vysokú vyhradenú hodnotu ukazujúcu mimo používanú pamäť. Vo vyhradenej časti pamäte budú uložené prázdne inštrukcie. Pôvodná hodnota programového čítača sa aj v tomto prípade uschováva a neskôr sa obnovuje. Čo je podstatné, hodnota čítača sa sleduje a ak sa zmení na hodnotu inú než je adresa nasledujúcej inštrukcie vo vyhradenej časti pamäte, program vykonal skok. Pokiaľ skok nastal, programový čítač je prepísaný na adresu skoku a program nie je zastavený. Pokiaľ skok nenastal, je po dokončení inštrukcie pred breakpointom do programového čítača uložená uschovaná hodnota. Program sa takto vykonáva podľa očakávania a nenadobúda neplatný stav ako v predchádzajúcom prípade. Riešenie nemá zásadný vplyv na rýchlosť simulácie. Ak nastane skok (nedôjde k zastaveniu na breakpointe), tak sa spomalenie rovná počtu čakacích cyklov. Predaním riadenia užívateľovi sa ale najpomalšou časťou simulácie stáva človek.

Pre reálne nasadenie do ladiaceho nástroja projektu Lissom bolo z vyššie popísaných dôvodov zvolené posledné spomenuté riešenie s popísanou úpravou.

5 TESTOVANIE

Testovanie riešenia možno rozdeliť na dve časti. Prvá časť testovania na jednoduchých modeloch procesorov na inštrukčnej úrovni mala zaistiť, aby sa implementovaním novej funkcionality nevyradila už existujúca. Testovalo sa na existujúcich modeloch dostupných v rámci projektu Lissom, *mips_basic* (http://www.mips.com/) a *vex* (http://www.hpl.hp.com/downloads/vex/). Druhá čas testovania prebiehala priamo na modeloch procesorov na úrovni cyklu s prúdovým spracovaním, modely *adop_ca* a *codea* (http://codasip.com/products/codea1.php). Toto malo odhaliť prípadné chyby v implementácii.

6 ZÁVER

Výsledkom tejto práce je funkčná implementácia ladiaceho nástroja, ktorý umožňuje pohodlné ladenie softwaru na simulátoroch na úrovni cyklu. Ďalej umožňuje prepínanie medzi inštrukčnou úrovňou a úrovňou cyklu. Funkčnosť implementácie bola overená na existujúcich modeloch procesorov dostupných v rámci projektu Lissom. Nová funkcionalita bola začlenená do projektu.

7 POĎAKOVANIE

Tento príspevok vznikol za podpory grantu MPO TIP FR-TI1/038, projektu Centra excelence IT4Innovations (CZ.1.05/1.1.00/02.0070) a výskumného zámeru MSM0021630528.

REFERENCE

[1] Rosenberg, J. B. How debuggers work: Algorithms, Data Structures, and Architecture. New York, NY, USA: John Wiley & Sons, Inc., 1996. ISBN 0-471-14966-7.

[2] Křoustek, J., Přikryl, Z., Kolář, D. et al. Retargetable Multi-level Debugging in HW/SW Codesign. In The 23rd International Conference on Microelectronics (ICM 2011). Hammamet, TN: Institute of Electrical and Electronics Engineers, 2011. S. 6. ISBN 978-1-4577-2209-7.

[3] Masařík, K. Systém pro souběžný návrh technického a programového vybavení počítačů. Brno, CZ: FIT VUT, 2008. 156 s. ISBN 978-80-214-3863-7.