Resumo do artigo "Architectural semantics for practical transactional memory"

Referência Bibliográfica : A. McDonald, et al., “Architectural semantics for practical transactional memory,” In Proc. 33Rd Int. Symp. on Computer Architecture, pp. 53–65 2006.

Autor do resumo : Luís Felipe Strano Moraes (RA: 016681)

O artigo, um dos mais recentes a trabalhar com memórias transacionais, mais especificamente com implementação em hardware, é interessante por propôr uma extensão ao ISA que permite funcionalidades mais ricas tanto nas linguagens de programação modernas quanto nos sistemas operacionais e bibliotecas disponíveis. O artigo começa com uma breve descrição do que é memória transacional e mencionando alguns dos principais problemas existentes impedindo o uso dela na prática. Em seguida, são propostos três mecanismos para resolver esses problemas, que serão descritos separadamente a seguir :

1) Two-phase transaction commit
Ao contrário dos sistemas propostos anteriormente aonde o commit ocorria pontualmente no final de uma transação, os autores sugerem a implementação de um commit em duas fases. Uma primeira instrução faria a validação da transação (chamada de xvalidate) e uma segunda (xcommit) efetuaria as escritas e tornaria os efeitos da transação visíveis ao exterior.
Isto permite que múltiplas transações trabalhando num mesmo tarefa possam coordenar seus commits, entre outras coisas.

2) Suporte a software handlers
São propostos 3 tipos diferentes de handlers :
a) Commit Handlers : permitem que funções específicas sejam executadas assim que uma transação faça o commit. Servem por exemplo para executar código com efeitos colaterais (como por exemplo escrever num arquivo).
b) Violation Handlers : permitem que funções específicas sejam executadas quando um conflito é detectado. O principal uso seria para permitir gerenciamento de contenção em software.
c) Abort Handlers : são similares aos de violação, porém são chamados somente quando uma transação é abortada explicitamente.

3) Closed- e Open-nested transactions
Ao contrário de várias implementações que tratam aninhamento de transações via flattening, é proposto o tratamento de transações aninhadas de duas formas :
a) Closed-nesting : a transação interna é vista como uma transação separada, porém as alterações que ela faz só se tornam visíveis ao exterior quando a transação mais externa faz o seu commit. Isso permite que um problema numa transação interna não faça com que a transação mais externa tenha que ser re-executada, e também permite um melhor gerenciamento de conflitos.
b) Open-nesting : a transação interna é vista como uma transação separada, e suas alterações ficam imediatamente visíveis ao exterior assim que ela terminar com sucesso. Elas podem ser usadas para fazer chamadas que atualizem o estado do sistema antes da transação mais externa terminar (e daí, caso seja necessário desfazer algo que elas fizeram no evento da transação mais externa não conseguir terminar com sucesso, pode-se usar os violation e abort handlers mencionados anteriormente.

Inicia-se então uma discussão dos aspectos práticos de uma implementação. O principal ponto levantado é a discussão sobre como tratar em hardware o aninhamento de transações. São propostas duas alternativas, uma que extende a linha da cache (porém é limitada a 4 níveis de aninhamento) e outra que usa diferentes linhas para diferentes níveis (sendo portanto mais extensível, porém sua implementação é mais complicada e a latência é maior).
O artigo termina com uma avaliação da performance da implementação que eles fizeram (extendendo um simulador de PowerPC) com uma série de benchmarks, mostrando a possível melhora de performance que uma implementação real desse sistema poderia obter (em torno de 2.2x nos testes do SPECjbb2000).