Analisando as Deficiências de Desempenho Apresentadas pelo Suporte Transacional do GCC

Bruno Chinelato Honorio[1], Alexandro Baldassin[1], João P. L. de Caravalho[2]

[1] Universidade Estadual Paulista (UNESP-DEMAC)
Rio Claro – SP – Brazil

[2]Universidade Estadual de Campinas (UNICAMP-IC)
Campinas – SP – Brazil

brunochonorio@gmail.com, alex@rc.unesp.br, joao.carvalho@ic.unicamp.br

Abstract. *The new concurrent programming mechanism, transactional memory, has received a lot of attention both from the academy and industry, which is evidenced by the support implemented in modern processors and compilers. Despite these efforts, transactional memory still is not broadly used, and one of the main reasons for that is the high computational cost that the code generated from transactional applications have shown. This work analyses the performance of the code generated by a widely used modern compiler, the GNU C Compiler (GCC), and presents, through extensive experiments, the reason behind the performance deficiencies of the transactional support of GCC.*

Resumo. *O novo mecanismo de programação concorrente, memória transacional, tem recebido grande atenção da academia e indústria, evidenciado pelo suporte implementado em processadores e compiladores contemporâneos. Apesar desses esforço, a memória transacional ainda não é amplamente adotada, e uma das principais razões é o alto custo computacional que o código gerado de aplicações paralelas tem mostrado. Este trabalho analisa o desempenho do código gerado por um compilador moderno amplamente usado, o GNU C Compiler (GCC), e apresenta, através de experimentos extensivos, a razão das deficiências de desempenho do suporte transacional do GCC.*

1. Introdução

Uma nova abordagem para o desenvolvimento de aplicações paralelas que tem sido bastante explorada é a memória transacional (*transactional memory* – TM) [Harris et al. 2010]. Esta abordagem tem recebido bastante atenção por oferecer um novo nível de abstração sobre os mecanismos tradicionais de sincronização em programação concorrente (criados nas décadas de 60 e 70), diminuindo assim a complexidade de programação. A memória transacional tem como principal conceito a transação: uma região de código que é executada atomicamente e de forma isolada. Desta forma, o programador precisa apenas se preocupar em explicitar o trecho de código que acessa regiões compartilhadas de memória, sendo então responsabilidade da implementação do sistema transacional garantir atomicidade e isolamento.

Para que o uso de memória transacional se torne ubíquo, é necessário que os compiladores mais utilizados possuam suporte transacional e que esse suporte ofereça um

desempenho equivalente ou superior à instrumentação manual. Um dos compiladores que oferece suporte transacional é o GCC (GNU C Compiler), desde a versão 4.7. Apesar do suporte oferecido, o código gerado pelo compilador tem apresentado custos muito altos, o que acaba desestimulando o uso abrangente de memória transacional.

O objetivo desse trabalho é analisar essa deficiência de desempenho apresentada pelo GCC e apresentar, através de vários testes realizados, o fenômeno que tem causado este *overhead* no suporte transacional do GCC.

O artigo é dividido da seguinte forma: A Seção 2 apresenta a descrição do problema, a Seção 3 descreve a metodologia empregada, e a Seção 4 apresenta os resultados obtidos e conclusões.

2. Descrição do Problema

O compilador, dado um programa escrito através de construções transacionais, deve gerar o código para iniciar/finalizar a transação, além de instrumentar todo o acesso às variáveis compartilhadas. O suporte para TM no GNU C Compiler (GCC) foi adicionado na versão 4.7. O suporte é considerado experimental, significando que a sua implementação ainda não está otimizada.

O funcionamento do suporte de TM no GCC pode ser dividido em três partes: declarações de transações, anotações de funções e as bibliotecas em tempo de execução. A declaração de transações refere-se ao bloco de código que é explicitado como a região onde ocorrerão as transações. A anotação de uma função define o nível de segurança que uma determinada função possui e como ela deve ser chamada dentro de uma transação. As bibliotecas em tempo de execução são as bibliotecas que são carregadas dinamicamente com o código compilado durante a execução, permitindo que o executável da aplicação seja carregado com diferentes bibliotecas transacionais com diferentes implementações sem a necessidade de recompilação.

Os problemas de desempenho do suporte transacional para o GCC tem origem em um aspecto em particular do código transacional: a adição de barreiras para leitura e escrita de variáveis compartilhadas dentro do bloco de código transacional. Em implementações em software, esse problema se agrava ainda mais, já que cada invocação necessita acessar metadados da transação para checar a consistência da execução. Com isso em consideração, o compilador deve a todo custo conseguir otimizar o código que gera, de forma a evitar ao máximo a instrumentação das leituras e escritas. Por exemplo, variáveis locais não necessitam de barreiras, já que seu escopo é local à transação. Infelizmente nem sempre é fácil tomar essa decisão em tempo de compilação, forçando o compilador a tomar uma atitude pessimista e insirir barreiras em váriaveis locais sempre que não seja possível decidir se o acesso é local ou global. Este fenômeno é chamado de instrumentação excessiva (*Overinstrumentation*).

3. Metodologia Experimental

Antes de conduzir os experimentos mais abrangentes, realizou-se um pequeno teste para evidenciar a ocorrência de instrumentação excessiva no código gerado pelo GCC. A Figura 1 mostra um exemplo simples de alocação local de memória dinâmica dentro de uma transação. A função `Foo` (linhas 1 a 8) cria uma transação (linha 2) que aloca memória

```
1  void Foo() {
2    __transaction_atomic {
3      list_t *pm = (list_t *) malloc(SIZE*sizeof(list_t));
4      // ...
5      Bar(pm);
6      // ...
7    }
8  }
9  static __attribute__((transaction_safe))
10 void Bar(list_t *p) {
11   while (p->next != NULL) {
12     int a = p->v;
13     // ...
14     p = p->next;
15   }
16 }
```

Figura 1. Exemplo hipotético com alocação local de memória dinâmica

```
1  BAR:
2    // ...
3    // int a = p->v;
4    400885: callq  4006b0 <_ITM_RU4>
5    // p = p->next;
6    // ...
7    400898: callq  4006a0 <_ITM_RU8>
8    // (p->next != NULL)
9    // ...
10   4008ac: callq  4006a0 <_ITM_RU8>
11   // ...
```

Figura 2. Código em linguagem de montagem gerado para o exemplo da Figura 1

dinâmica (linha 3) e que após algum processamento, é passada como parâmetro para uma função chamada `Bar` (linha 5). Como o trecho de memória alocado serve apenas para processamento local à transação, não é necessário que ele seja instrumentado. A função `Bar` (linhas 10-16) realiza uma operação qualquer com a lista ligada através do laço criado (linhas 11-15). Apesar da declaração da lista ligada `pm` ser local à função `Foo`, levando à conclusão que esta lista ligada não seria instrumentada pelo compilador, a Figura 2 apresenta o problema de instrumentação em excesso esperado.

A Figura 2 mostra, de maneira bastante simplificada, trechos do código em linguagem de montagem gerados pelo compilador para o código da Figura 1. Nesta Figura, apenas as operações transacionais da função `Bar` foram destacadas. Todas as operações do laço que envolvem trabalhar com a lista ligada `p`, declarada localmente na função `Foo`, invocaram operações transacionais. Operações transacionais seguem uma nomenclatura específica definida pela documentação da interface binária de aplicação (ABI) da Intel [int 2008]. As operações denotadas (linhas 4, 7 e 10) especificam barreiras de leitura usadas sobre os acessos ao elemento da lista ligada (linha 4), ao próximo ponteiro (linha 7), além do próximo elemento encadeado (linha 10).

Este exemplo mostra que, de fato, o compilador instrumenta todo o acesso à lista ligada. Esse fato implica que como todo acesso realiza uma chamada a um procedimento da biblioteca transacional, o desempenho desse código deve ser muito pior que o de um código sem essa instrumentação excessiva.

4. Resultados Experimentais e Conclusões

Os experimentos apresentados a seguir objetivam comparar o desempenho do código gerado pelo GCC com a versão manualmente instrumentada das transações das aplicações

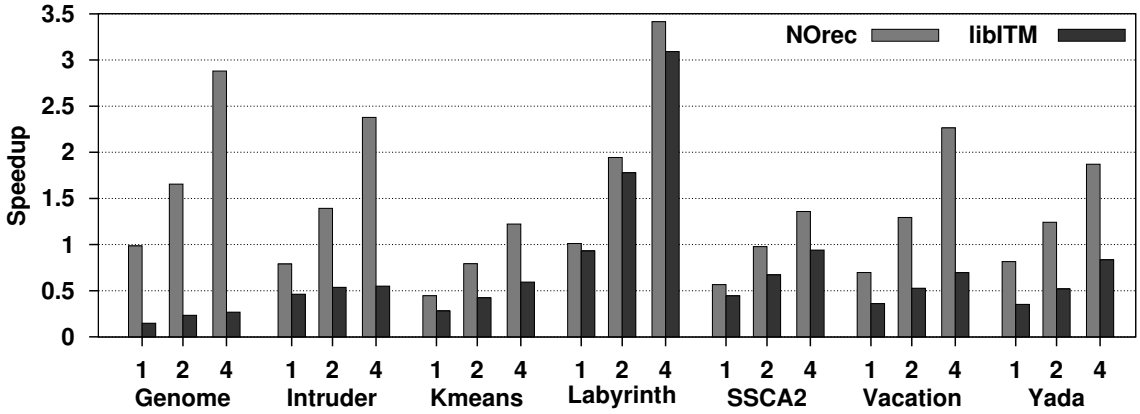

Figura 3. Comparação de desempenho entre a versão instrumentada manualmente (`NOrec`) e a gerada pelo compilador GCC (`libITM`). Ambas usam a STM conhecida como NOrec.

do pacote STAMP [Minh et al. 2008]. Ambas as versões utilizam como biblioteca em tempo de execução a STM conhecida como NOrec[Dalessandro et al. 2010], fazendo com que as comparações sejam justas.

Os experimentos foram realizados em uma máquina servidor com versão do Linux 4.8.6, processador *Intel®Core™* I7 4770 (8 núcleos: 4 físicos + 4 lógicos) com 16GB de memória RAM e compilador GCC versão 5.1. Os resultados reportados representam a média de 20 execuções para 1, 2 e 4 threads.

A Figura 3 apresenta os resultados obtidos. `libITM` refere-se à versão gerada pelo compilador e `NOrec` refere-se à versão instrumentada manualmente. Nota-se pela figura que a versão instrumentada manualmente é melhor que a gerada pelo compilador em todos os casos, particularmente para as aplicações `Genome, Intruder, Yada`. Esse desempenho inferior, como argumentado anteriormente, é por conta da instrumentação excessiva causada pelo compilador. O objetivo de trabalhos futuros é guiar o compilador, através de anotações fornecidas pelo programador, a não instrumentar variáveis locais para evitar a ocorrência de instrumentação em excesso e assim melhorar o desempenho do código gerado pelo compilador de aplicações transacionais.

Referências

(2008). *Intel Transactional Memory Compiler and Runtime Application Binary Interface - Revision 1.0.1*. Intel Corporation.

Dalessandro, L., Spear, M. F., and Scott, M. L. (2010). NOrec: Streamlining STM by abolishing ownership records. In *Proceedings of the 15th Symposium on Principles and Practice of Parallel Programming*, pages 67–78.

Harris, T., Larus, J., and Rajwar, R. (2010). *Transactional Memory*. Morgan & Claypool Publishers, 2 edition.

Minh, C. C., Chung, J., Kozyrakis, C., and Olukotun, K. (2008). STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 35–46.