Redução de Falhas através da Reordenação do Código

Petterson Luís de L. S. Vieira e Edil S. T. Fernandes
Universidade Federal do Rio de Janeiro
Programa de Engenharia de Sistemas e Computação, COPPE
Caixa Postal 68511
21941-972 Rio de Janeiro - RJ, Brazil
plvieira, edil@cos.ufrj.br

Resumo

A diferença de velocidade entre a CPU e o sistema de memória tem aumentado continuamente nos últimos anos provocando considerável queda no desempenho dos processadores. Uma alternativa para reduzir essa queda consiste em organizar as instruções do código binário de modo que as falhas na "cache" de instruções sejam também reduzidas.

Este artigo descreve a condução de experimentos para determinar, independentemente da entrada de dados, quais foram as instruções que sempre foram executadas e quais as que permanceram intocadas durante a integral execução dos programas inteiros do conjunto SPEC95.

Os resultados produzidos por nossos experimentos são valiosos pois eles permitem posicionar inicialmente as instruções de um programa que será executado segundo um esquema que realiza o posicionamento de suas instruções dinamicamente. Dessa forma, a taxa de falhas na "cache" de instruções será reduzida e conseqüentemente, o desempenho do processador será aumentado.

1. Introdução

A velocidade dos processadores supera a dos sistemas de memória e essa diferença continua a aumentar consistentemente: a cada ano, a velocidade dos processadores é aumentada 60% e a das das memórias 10% [7].

O constante avanço na tecnologia de fabricao de circuitos VLSI (*Very Large Scale of Integration*) já acena com o aparecimento de processadores capazes de atingir freqüências de até 10 GHz em poucos anos [6]. Conhecida como *memory gap problem*, essa diferença de velocidades provoca longas interrupções no processamento sempre que ocorre uma falha na memória *cache* de instruções. Por exemplo, conforme destacado por Wilkes em [7], numa estação de trabalho Alpha 21264 com 500 MHz de freqüência, cada falha na *cache* de instruções faz com que o processador permaneça interrompido durante 128 ciclos do *clock*, aguardando pela chegada da instrução faltosa. Esse período seria de 256 ciclos se a velocidade da memória permanecesse a mesma e a velocidade do processador passasse para um GHz (devemos observar que já temos, agora no primeiro semestre de 2002, processadores cujas freqüências ultrapassaram dois GHz).

Se uma solução não for encontrada, o problema da diferença entre as velocidades será agravado ainda mais. Para reduzir o nmero de falhas na "*cache*" de instrues e conseqüentemente o número de ciclos que o processador permanecer interrompido aguardando pela chegada da próxima instrução, algumas técnicas de rearranjo do código têm sido apresentadas na literatura (*code layout* ou *code reordering techinques*).

Pettis e Hansen apresentaram em 1990 algumas técnicas para o posicionamento de *procedures* e de blocos básicos de um programa [4]. As técnicas desenvolvidas pelos autores são baseadas no perfil de execução dos programas e foram empregadas nos compiladores da arquitetura PA-RISC da HP. Os tempos de processamento dos programas reordenados, apresentaram ganhos médios variando de 8 a 10%.

Mais recentemente, pesquisadores da Compaq e da Universidade de Catalunha apresentaram um estudo sobre o efeito de técnicas de posicionamento de código em aplicações do tipo "Processamento de Transações." Os autores verificaram que com o uso dessas técnicas, as taxas de falhas na *cache* de instruções foram reduzidas de até 65% e que o tempo de processamento da bateria de programas de teste ficou 1,33 vezes melhor do que originalmente [5].

A técnica de reordenação dos blocos básicos de um programa pode ser assim descrita: (i) toda vez que a posição de um bloco muda, é necessário movimentar para cima os outros blocos que o sucediam, compactando desse modo, o código; (ii) em seguida, precisamos garantir que os blocos ancestrais continuem acessando os blocos que foram movi-

mentados (se ajustarmos o endereço alvo em cada ancestral, o bloco que mudou de posição, continua acessível); (iii) se a última instrução do bloco ancestral não for uma transferência de controle (e se ele não for contíguo ao bloco movimentado) então, uma instrução (extra) de desvio incondicional deverá ser introduzida no final do bloco ancestral; (iv) todos os outros endereços alvo das instruções de desvios também devem ser ajustados se o bloco destino foi movimentado (pela compactação).

Em trabalhos anteriores [2–3] verificamos que a percentagem de blocos básicos (e de instruções) dos programa inteiros do conjunto SPEC95 que nunca são executados é significativa, chegando geralmente a ultrapassar metade do código binário. Apesar de nunca executados, estes blocos básicos não podem ser simplesmente retirados do código, pois eles são responsáveis pelo tratamento de exceções que podem surgir durante o processamento do programa.

Uma vez que já sabemos que a utilização dos blocos básicos de um programa é modesta, então seria muito útil manter o trecho do programa que sempre é executado, separado dos blocos básicos que nunca serão executados. Com essa estratégia, evitaremos que uma mesma linha da *cache* de instruções seja compartilhada por blocos usados e não usados. Desse modo, garantiremos que os blocos básicos que nunca serão executados, ficarão armazenados nos níveis inferiores da hierarquia do sistema de memória. Conseqüentemente, haverá mais espaço na *cache* para o armazenamento daquelas instruções que serão executadas de fato.

O objetivo desse estudo não é a reordenação de código propriamente dita, mas sim, fornecer os meios para a investigação e implementação de técnicas que reordenam o código. Conforme veremos em seguida, além de fornecer sugestões para a movimentação dos blocos básicos, nossos experimentos revelaram surpreendentes detalhes sobre a utilização dos blocos básicos (e instruções) de um programa.

Este artigo descreve os experimentos que realizamos para identificar e quantificar aqueles blocos básicos de um programa que:

- sempre foram executados e;
- os blocos que permaneceram intocados.

Quando mencionamos que um bloco sempre foi executado (ou que ele permaneceu intocado), estamos querendo dizer que esse fato ocorreu após examinar seu comportamento ao longo de diversas execuções do respectivo programa. Isto foi feito para todos blocos de cada programa da bateria de teste.

Nosso processo de identificação e quantificação dos blocos básicos é realizado somente após diversas execuções de um mesmo programa. Cada execução difere das outras pelo conjunto de dados de entrada que foi processado.

Essa identificação é muito útil pois ela indica como os blocos básicos devem ser rearranjados no programa executável, gerando assim um código mais eficiente que produzirá um número menor de falhas na *cache* de instrues e que portanto, será executado mais rapidamente.

Este trabalho está organizado em cinco seções. A Seção 2 apresenta nosso ambiente experimental: a arquitetura usada, a bateria de programas de teste e os cojuntos de dados de entrada dos programas de teste. A Seção 3 descreve os experimentos e a Seção 4 apresenta e discute os resultados obtidos. A Seção 5 fornece as principais conclusões do trabalho

2. Ambiente experimental

Um bloco básico é uma importante estrutura em Computação: blocos básicos são seqüências de instruções que apresentam a seguinte propriedade – "se uma instrução do bloco básico for executada então, todas as outras também serão" (ou seja, blocos básicos são executados integralmente). Programas são formados por uma coleção de blocos básicos que devem ser executados seqüencialmente (um bloco após o outro), até que ocorra uma transferência de controle para um outro bloco não adjacente.

Toda vez que um programa é compilado / montado / ligado, seus blocos básicos são acomodados seqüencialmente no código de máquina numa ordem que não leva em consideração se o bloco será executado ou não. A Figura 1 ilustra como os blocos básicos de um programa são acomodados no código binário e na memória principal.

Na Figura 1, os pequenos retângulos representam os blocos básicos de um programa. Estamos supondo que os retângulos brancos são os blocos básicos que permanecerão intocados durante toda a execução e que os outros são os blocos que serão executados pelo menos uma vez.

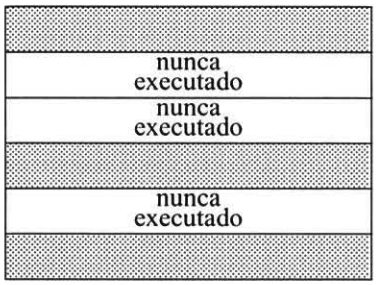

Figura 1. Diagrama de um programa objeto na memória

Essa mistura de blocos nunca executados com os outros, é responsável por uma queda substancial no desempenho de processadores. Por exemplo, uma linha da *cache* de instruções armazenando quatro instruções, pode conter

instruções provenientes de até quatro blocos distintos. Se duas destas instruções pertencerem a dois blocos que permanecerão intocados, então o desperdício nesta linha é de 50%. Para agravar ainda mais a situação, falhas (em outras linhas) irão ocorrer e melhor seria se as instruções de blocos não executados permanecessem em um nível mais inferior da hierarquia de memória.

Na realidade, este é o objetivo final de nosso estudo: manter os blocos básicos que serão usados, separados dos demais. O arranjo dos blocos na memória que estamos interessados em produzir é apresentado na Figura 2.

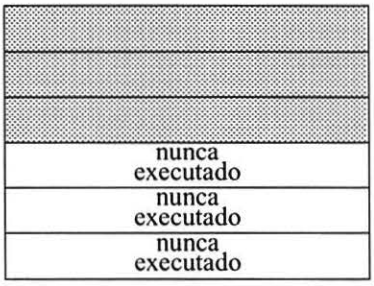

Figura 2. Novo arranjo dos blocos do programa objeto

Os programas empregados como teste (*benchmarks*) são provenientes do conjunto SPEC95. Os oito programs inteiros deste conjunto (Go, M88ksim, Gcc, Compress, Li, Ijpeg, Perl e Vortex) foram simulados durante nossa investigação.

Adotamos a versão da arquitetura MIPS definida por Doug Burger e Todd M. Austin nos nossos experimentos. Os programas do SPEC95 foram estaticamente compilados (i.e., não há *Dynamic Loaded Libraries*) para a versão da arquitetura MIPS e distribuídos pelos autores do projeto *SimpleScalar* [1].

Cada programa de teste foi executado várias vezes: todos arquivos de entrada de dados dos conjuntos *test, training* e *reference* do SPEC95, foram usados na nossa investigação.

3. Monitorando a execução de programas

Considerando que estamos interessados em separar os blocos intocados dos outros, então precisamos monitorar o comportamento de cada bloco em tempo de execução. Então, empregamos um modelo desenvolvido anteriormente (BBMM – *a Basic Block Machine Model*). Este modelo trata cada bloco básico como sendo a unidade padrão de processamento: ao invés de processar uma única instrução por ciclo, nosso modelo processa integralmente um bloco básico por ciclo. O ciclo da máquina BBM é fragmentado em diversos subciclos, cada um deles processando uma instrução do bloco básico [2].

Precedendo a execução de um programa no nosso modelo, o código é examinado, seus blocos básicos são identificados e uma estrutura de dados descrevendo os blocos é produzida. Essa estrutura em conjunto com o programa binário (arquivo do tipo .ss) e com os dados de entrada (um arquivo dos conjuntos *test, training* ou *reference*), são processados pelo nosso modelo de máquina. Nosso processador de blocos básicos gera um arquivo (.per) contendo o perfil de execução de cada bloco básico.

Uma vez que o repertório de instruções reconhecido pelo nosso modelo é o da versão MIPS do *SimpleScalar*, então usamos o simulador "sim-fast" do pacote para processar as instruções de cada bloco básico.

Em tempo de execução, nosso modelo passa para o simulador "sim-fast" as instruções do bloco básico corrente e quando este for esgotado, o modelo verifica qual o endereço do próximo bloco que foi gerado pelo "sim-fast." Novamente, nosso modelo passa para o simulador, as instruções do bloco sucessor e assim por diante. Antes da execução do próximo bloco, o processador de blocos básicos incrementa o número de vezes que o bloco foi ativado e por esse motivo, no final do processamento do programa (.ss) teremos o número de vezes que um bloco foi executado. A Figura 3 mostra como nosso modelo de execução foi implementado.

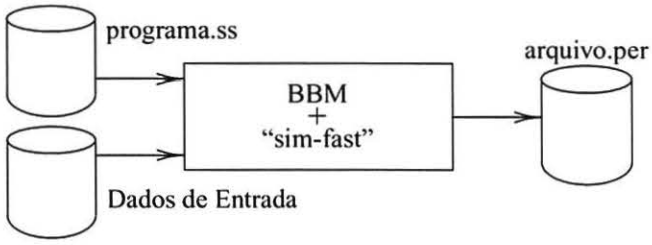

Figura 3. Implementação do modelo

O arquivo (.per) contém o resultado de uma única simulação do programa (.ss). Esse arquivo armazena o perfil de execução dos blocos básicos do programa quando do processamento de um determinado arquivo de dados de entrada. Mudando o arquivo de dados de entrada, outros caminhos poderão ser percorridos pelo fluxo de controle e por essa razão, teremos um outro perfil de execução dos blocos básicos e um outro total de instruções executadas.

Em outras palavras, apesar de já proporcionar um melhor desempenho, a nova ordenação dos blocos baseada em um único perfil de execução, não garante que ela também será eficiente no processamento de outros arquivos de dados de entrada. A busca por uma técnica de arranjo mais eficiente motivou os experimentos descritos a seguir na Seção 4.

4. Experimentos e seus resultados

Uma crítica relacionada com aquelas técnicas de reordenação de código (*code reordering techniques*) baseadas no perfil de execução, é que **talvez** elas não apresentem o mesmo desempenho se os dados de entrada forem modificados.

Se uma reordenação de código baseada em um único perfil de execução será eficiente (ou não) quando do processamento de outros arquivos de entrada, nada podemos afirmar: não realizamos este experimento e desconhecemos se ele foi já conduzido por alguém. Conseqüentemente, não estamos preparado para tecer comentários.

Conforme mencionado, ao invés de processar um único arquivo de dados de entrada, nossos experimentos com o modelo BBMM envolveram **todos** arquivos de entrada de dados do conjunto SPEC95. Para cada arquivo e cada programa, um diferente arquivo contendo o perfil de execução dos blocos básicos foi gerado. Nos experimentos apresentados em um outro trabalho e que revelaram que mais de 50% dos blocos básicos de um mesmo programa permanecem intocados [2–3], verificamos que este fato também é verdadeiro para cada um dos arquivos de dados de entrada individualmente (contidos nos conjuntos *test, training* e *reference* do SPEC95.

Tabela 1. Instruções executadas pelo Gcc

Entrada	Inst. Exec.	BBs
ref/ amptjp.i	1.276.630.247	78.951
ref/ c-decl-s.i	1.276.712.103	78.951
ref/ cccp.i	1.263.333.473	78.951
ref/ cp-decl.i	1.440.772.369	78.951
ref/ dbxout.i	168.130.341	78.951
ref/ explow.i	218.543.860	78.951
ref/ expr.i	1.022.510.261	78.951
ref/ insn-recog.i	563.975.402	78.951
ref/ integrate.i	222.314.032	78.951
ref/ protoize.i	508.129.155	78.951
ref/ recog.i	281.539.848	78.951
ref/ reload1.i	930.402.835	78.951
ref/ stmt.i	548.588.549	78.951
ref/ stmt-protoize.i	1.022.473.561	78.951
ref/ toplev.i	428.261.083	78.951
ref/ varasm.i	250.947.348	78.951

Nesta série de experimentos decidimos investigar qual seria o comportamento dos blocos se os arquivos contendo o perfil de execução (arquivos .per) fossem combinados.

Assim, geramos os arquivos (.per) de cada programa (no caso do programa Gcc, 16 execuções distintas foram realizadas, gerando 16 arquivos .per). As Tabelas 1 e 2 apresentam os resultados de cada execução do "Gcc" (a limitação

Tabela 2. Perfil de execução do Gcc

Entrada	Nunca	Executados
ref/ amptjp.i	44.343	34.608
ref/ c-decl-s.i	44.343	34.608
ref/ cccp.i	44.587	34.364
ref/ cp-decl.i	45.558	33.393
ref/ dbxout.i	51.854	27.097
ref/ explow.i	55.329	23.622
ref/ expr.i	45.553	33.398
ref/ insn-recog.i	55.732	23.219
ref/ integrate.i	48.693	30.258
ref/ protoize.i	46.716	32.235
ref/ recog.i	49.431	29.520
ref/ reload1.i	44.294	34.657
ref/ stmt.i	46.403	32.548
ref/ stmt-protoize.i	44.160	34.791
ref/ toplev.i	49.190	29.761
ref/ varasm.i	46.596	32.355

no número de páginas impediu a apresentação dos outros programas). Cada linha destas tabelas representa uma simulação específica. Na Tabela 1, a primeira coluna indica o conjunto de dados de entrada, a segunda coluna contém o total de instruções executadas e a terceira indica quantos blocos básicos formam o Gcc. A Tabela 2 lista os totais de blocos básicos do "Gcc" que permanceram intocados e que foram executados.

Em seguida, verificamos quais foram os blocos básicos que permaneceram intocados durante todas execuções de um mesmo programa. Em outras palavras, verificamos se um mesmo bloco básico permaneceu intocado em todos arquivos (.per) que foram gerados.

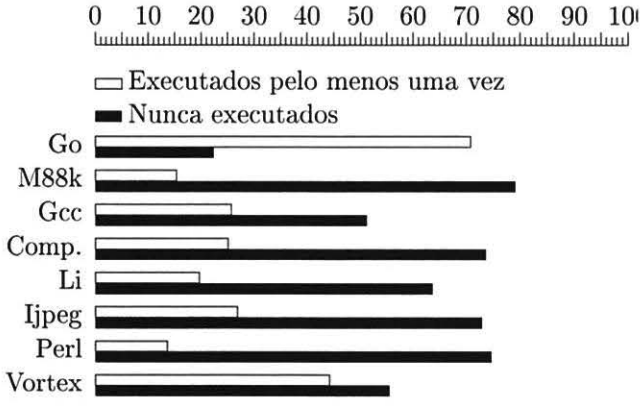

Figura 4. Utilização dos blocos básicos (%)

A Figura 4 apresenta para cada programa, a percentagem de blocos básicos que permaneceram intocados durante todas as execuções, junto com a taxa de blocos que foram

executados (pelo menos uma vez) durante o processamento de cada arquivo de dados de entrada.

As barras da Figura 4 representam as percentagens de blocos que nunca/sempre foram executados. A Tabela 3 apresenta um resumo geral dos experimentos envolvendo os blocos básicos. Examinando as percentagens da Figura 4, podemos verificar que o comportamento das execuções do programa "Go" é semelhante ao que ocorreu em experimentos anteriores [1]: este programa é bem otimizado e por esse motivo, o total de blocos básicos que permaneceram intocados é bem modesto quando comparado com os outros programas inteiros do conjunto SPEC95.

Para os demais programas, a percentagem de blocos intocados ultrapassa a marca dos 50% (percentagens variando de 51 a 79%). Esses valores são surpreendentes pois intuitivamente, era de se esperar percentagens bem inferiores.

Esta elevada taxa de blocos nunca usados é bastante promissora para o desenvolvimento de técnicas de reordenação, pois poderemos grupar um grande número de blocos básicos numa das extremidades do código objeto, forçando deste modo, que eles não sejam trazidos indevidamente para a memória *cache* de instruções.

Tabela 3. Resultados dos programas

Prog.	Inst. Exec.	BBs	Nunca	Exec
go	101.677.625.668	14.576	3.257	10.326
m88k	69.245.327.921	9.235	7.308	1.420
gcc	11.423.264.467	78.951	40.416	20.298
comp.	43.104.411.720	3.500	2.577	879
li	54.126.024.091	6.164	3.922	1.212
ijpeg	87.648.014.201	10.654	7.770	2.859
perl	39.073.666.980	19.206	14.341	2.611
vortex	85.586.137.244	28.382	15.751	12.551

A Tabela 3 condensa os resultados de todas as execuções. Os valores aqui listados referem-se ao total de instruções simuladas em conjunto com o número de blocos básicos que permaneceram intocados (durante as diversas simulações de um mesmo programa) e o total de blocos que sempre foram executados (pelo menos uma vez).

Podemos observar na Figura 4 que a soma das duas percentagens (dos blocos sempre/nunca executados) é inferior a 100%. Analogamente, a Tabela 3 indica que a soma dos blocos nunca/sempre executados é inferior ao total de blocos básicos do programa correspondente. Isto ocorre porque alguns blocos são executados somente no processamento de alguns arquivos de dados de entrada. Para os outros arquivos, eles permaneceram intocados. Neste caso, tais blocos não se enquadram como sendo dos tipos "nunca" ou "sempre" executados. No diagrama da Figura 5, estes blocos foram classificados como sendo do tipo "executados algumas vezes."

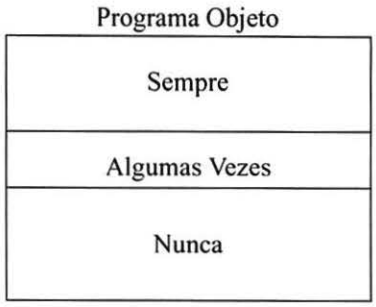

Figura 5. Classificação dos blocos básicos

A classificação apresentada na Figura 5 pode ser empregada por técnicas de reordenação de código. A porção do código incluindo os blocos que foram executados somente algumas vezes proporciona muitos oportunidades de investigação: como deveríamos ordenar os blocos desta região? Devemos iniciar o posicionamento com os blocos que foram executados mais vezes? Ou seria melhor realizar o posicionamento dinamicamente (i.e., durante a execução do programa objeto)? Estas e outras alternativas são excelentes tópicos de pesquisa.

5. Conclusões

Neste artigo apresentamos um estudo relacionado com o comportamento dos blocos básicos dos programas inteiros do conjunto SPEC95. Esse estudo foi bastante abrangente: ele envolveu o processamento de todos arquivos de entrada de dados do SPEC95, ou seja, para cada conjunto de entradas, realizamos uma nova execução do programa e anotamos a utilização de seus blocos básicos. No final, verificamos que bilhões de instruções foram executadas por cada programa: os resultados aqui divulgados resultam desse total de instruções.

Conforme havíamos observado anteriormente, os experimentos aqui apresentados novamente comprovaram que a maioria dos blocos básicos de um programa **nunca são executados**. Ao modificar o conjunto de dados de entrada, o programa que está sendo investigado apresentará uma diferente contagem de blocos (e instruções) executados, já que o caminho percorrido pelo fluxo de controle será diferente.

Por outro lado, não havia nenhuma garantia que a percentagem de blocos básicos que permaneceriam intocados com o novo conjuntos de dados de entrada, ainda continuaria ultrapassando a metade. Se considerarmos que nossa contagem de blocos intocados apenas inclui aqueles blocos que não foram ativados por nenhuma execução do programa, e à medida que o número de execuções distintas cresce, a tendência seria obter uma percentagem de blocos intocados menor que 50%. Por esse motivo é que essa percentagem surpreendeu nossa expectativa.

O comportamento observado dos blocos básicos indica que o tempo de processamento dos programas pode ser reduzido consideravelmente se o *code layout* for otimizado. A reordenação criteriosa dos blocos básicos (por exemplo, mantendo separado os que nunca foram utilizados dos outros), permitirá que a memória *cache* de instrues no contenha instruções que nunca serão executadas. Conseqüentemente, haverá mais espao para as instruções que serão realmente utilizadas, reduzindo desse modo o número de falhas na memória *cache* de instruções, o que resulta num tempo de processamento menor.

Nosso estudo representa uma pequena contribuição para aliviar os efeitos produzidos pela diferença de velocidades entre a CPU e o sistema de memória (*the memory gap problem*). Utilizando a combinação dos perfis de execução dos blocos básicos de um programa, novas técnicas de reordenação de código podem ser desenvolvidas. Nossos trabalhos nesta direção prosseguem.

6. Referências

[1] Doug Burger and Todd M. Austin, "The Simplecalar Tool Set, Version 2.0", Technical Report #1342, Computer Sciences Department, University of Wisconsin–Madison, June 1997, pp.1–21.

[2] Edil S. T. Fernandes e Gabriel P. Silva, "BBM — Um Processador de Blocos Básicos", Proceedings of the WSCAD'00 - 1st Workshop on High Performance Computing Systems, S. Pedro, SP, Brazil, October 2000, pp. 3–8.

[3] Edil S. T. Fernandes and Valmir C. Barbosa, "Monitoring the Structure and Behavior of Programs," Proceedings of the 4th International Conference on Massively Parallel Computing Systems, MPCS'02, Ischia, Italy, ISBN: 0-9669530-0-2, Published by the National Technological University Press, 10–12/April/2002, 6 pag

[4] Karl Pettis and Robert C. Hansen, "Profile Guided Code Positioning," Proceedings of the Conference on Programming Language Design and Implementation, White Plains, NY, June 1990, pp. 16–27.

[5] Alex Ramirez, Luiz A. Barroso, Kourosh Gharachorloo, Robert Cohn, Josep Larriba-Pey, P. Geoffrey Lowney, and Mateo Valero, "Code Layout Optimizations for Transaction Processing Workloads," Proceedings of the 28th Annual International Symposium on Computer Architecture, June 2001, pp. 155 – 164.

[6] André Seznec, "Hiding Thousand Cycles Memory Latency Through L2 Prefetching," in http://www.irisa.fr/theses2001/caps6.htm

[7] Maurice V. Wilkes, "The Memory Gap," Keynote address of the Workshop in the Memory Wall Problem, June 2000, in http://www.ece.neu.edu/conf/wall2k/wilkes1.pdf