BBM - um Processador de Blocos Básicos

Edil S. T. Fernandes e Gabriel P. Silva
Universidade Federal do Rio de Janeiro
e-mail: {edil,gabriel}@cos.ufrj.br

Resumo

No artigo "empirical study of FORTRAN programs" Donald Knuth observou que 4% das instruções eram responsáveis por 50% do tempo de execução de um programa. Um outro fato relacionado com o código objeto, é que 10% de suas instruções são responsáveis por 90% das instruções executadas. Neste artigo nós avançamos um pouco mais: durante nossos estudos com BBM –um processador que executa blocos básicos– percebemos que quando da geração de um programa objeto, o futuro de uma grande parte de seus blocos básicos já está decidida: eles nunca serão executados.

Há algumas razões para este comportamento: a generalidade das bibliotecas dos sistemas operacionais que são agregadas ao programa de aplicação; o estilo defensivo do programador da aplicação; o conjunto de dados de entrada, e assim por diante.

Embora ignorado pelos projetistas de hardware/software esta característica dos programas objetos possui diversas implicações na organização e desempenho dos processadores atuais: cache de instruções e fill units seriam muito mais eficientes se o processo de mapeamento das instruções nas caches levasse em conta as fronteiras de cada bloco básico e a sua freqüência de utilização.

O artigo descreve nossos experimentos com BBM. Durante a execução dos programas do SPECint95, verificamos que para a maioria destes programas mais do que 50% dos blocos básicos nunca são utilizados.

1. Introdução

No modelo clássico de computação, uma única instrução é a unidade padrão de processamento: apenas uma instrução é buscada, decodificada e despachada a cada ciclo de processador. Com o advento das arquiteturas super escalares e VLIW [FISH84], temos um outro modelo no qual mais de uma instrução pode ser buscada, decodificada e despachada por ciclo de processador. Explorando o paralelismo a nível de instrução (ILP) dos programas, estas arquiteturas são uma tendência quando do projeto de novas máquinas.

Processadores super escalares recentes podem despachar diversas instruções por ciclo. Os processadores Sun UltraSPARC, MIPS R10000, Alpha 21164 e AMD K5 são exemplos de máquinas que podem despachar até quatro instruções por ciclo.

Apesar das altas taxas de acerto dos mecanismos atuais de predição de desvios, as unidades de busca são incapazes de extrair múltiplas –e desalinhadas– instruções da memória *cache*. Conseqüentemente, ao invés de quatro instruções por ciclo, é comum encontrarmos menos que duas instruções por ciclo sendo concluídas nesses processadores.

Nos anos recentes há um crescente interesse em aumentar o número de instruções processadas em paralelo. Um dos principais problemas é a eficiência da unidade de busca. Os trabalhos conduzidos por Y. Patt ([SPFP97] e [SPEP98]), J. E. Smith [ERBS96] e A. Seznec [SJSM96] são exemplos do esforço para aumentar a eficiência das unidades de busca.

Em 1971, no seu famoso "empirical study of FORTRAN programs," Donald Knuth verificou que 4% das instruções de um programa são responsáveis por 50% do tempo de execução [KNUT71]. O perfil de utilização das instruções, fato longamente reconhecido, mostra que 10% das instruções de um programa são responsáveis por 90% das instruções executadas [JHDP90].

Durante os experimentos realizados com uma máquina hipotética orientada para a execução de blocos básicos (BBM), descobrimos resultados bastante relevantes quanto ao comportamento dos blocos básicos: mais que 50% dos blocos básicos de um programa nunca são utilizados.

Em um trabalho anterior, Y. Patt e seu grupo apresentou o conceito de uma arquitetura estruturada em blocos básicos [SMSP88]. Na ocasião, eles estavam investigando técnicas para a formação de blocos expandidos e para a predição de desvios para essa arquitetura. Também estamos interessados no conceito arquitetural de um processador de blocos básicos. Contudo, nosso principal objetivo é apresentar o perfil do comportamento dos blocos básicos.

O artigo está organizado em cinco seções. A seção seguinte apresenta um resumo dos conceitos explorados na concepção da máquina BBM. A Seção 3 descreve como os experimentos foram realizados, quais foram as ferramentas de simulação, os programas de avaliação, e o conjunto de dados utilizados no estudo. A Seção 4 mostra o resultado dos experimentos e na Seção 5 apresentamos as principais conclusões.

2. Conceitos Básicos

Como ocorre com as arquiteturas microprogramadas com microinstruções horizontais –que são as ancestrais dos processadores VLIW da atualidade– um grupo de instruções é a unidade padrão de processamento da BBM. Este grupo de instruções forma um bloco básico que é assim definido:

Definição: *Um bloco básico é uma coleção ordenada de instruções de máquina sem pontos de entrada, exceto a primeira instrução do bloco, e sem desvios, exceto, possivelmente, a última instrução do bloco* [LDSM80].

Segundo essa definição, é fácil verificar que toda vez que um bloco básico é selecionado para a execução (i.e., quando ele é ativado), todas as

suas instruções serão executadas. Já que o bloco básico será executado integralmente, seria vantajoso manter os blocos básicos separados. A separação (e armazenamento em estruturas de *hardware*, por exemplo uma memória *cache*), de blocos básicos torna o mecanismo de busca de instruções mais simples e mais eficiente, sendo desnecessário alinhar, mascarar e concatenar as instruções conforme requerido por aquelas arquiteturas que despacham diversas instruções em paralelo [CMMP95]. Podemos fazer uma analogia entre o conceito de memória virtual e a vantagem no armazenamento separado dos blocos básicos.

A implementação de *caches* de instrução, emprega uma técnica similar aos sistemas com memória virtual que empregam a paginação [DENN70]. Na paginação, é possível encontrar trechos mutuamente exclusivos de um programa compartilhando a mesma página da memória física. Conseqüentemente, ocorre um desperdício no espaço de armazenamento.

Similarmente, na *cache* de instruções, é comum encontrarmos instruções pertencentes a diferentes blocos básicos na mesma linha da *cache*. Levando em conta que blocos que são espacialmente vizinhos, nem sempre são vizinhos temporalmente, então seria melhor manter estes blocos separadamente. Este é o argumento utilizado em favor da técnica da segmentação pelos seus defensores.

Para ilustrar as dificuldades encontradas em processadores super escalares que despacham diversas instruções por ciclo, um exemplo é dado a seguir. A Figura 1 apresenta duas linhas adjacentes da *cache* contendo instruções dos blocos básicos a, b e c.

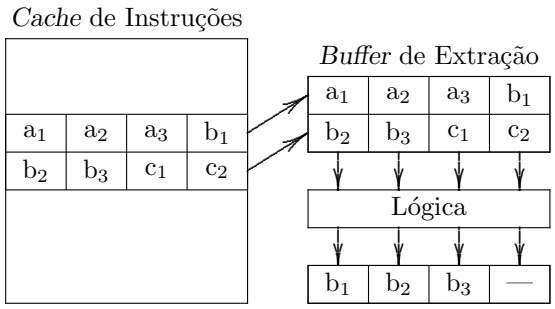

Figura 1: Extração do Próximo Bloco

Estamos supondo que o fluxo de controle é transferido para b_1 (a última instrução da primeira linha da *cache*) e que a unidade de preenchimento (*fill unit*) pode transferir até quatro instruções para o estágio de decodificação. Por esta razão é necessário ler o conteúdo das duas linhas contíguas (assumindo que a *cache* permita a leitura de duas linhas em paralelo). O conteúdo das duas linhas é transferido para o *buffer* de extração representado na Figura 1. Inicialmente, é necessário remover as instruções pertencentes aos blocos a e c. Em seguida, é preciso modificar a posição das instruções dentro do bloco básico b: b_1 é a última instrução da linha e deve ser a primeira instrução que será passada para o estágio de decodificação; por esse motivo, as instruções b_2 e b_3 devem ser deslocadas de uma posição para a direita. A lógica responsável por esta tarefa é bastante complexa. Se a *cache* de instruções estivesse organizada em termos de blocos básicos, estas tarefas se tornariam menos complexas ou desnecessárias. A Figura 2 ilustra uma memória *cache* orientada para o armazenamento de blocos básicos.

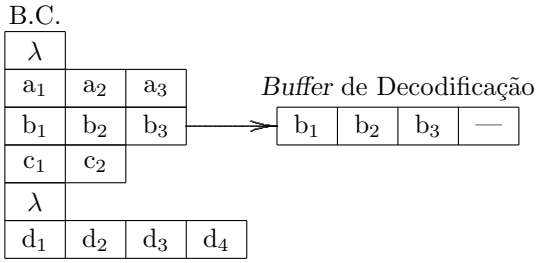

Figura 2: *Cache* de Blocos Básicos

A Figura 2 mostra seis linhas de uma *cache* que armazena alguns blocos básicos. Há duas entradas vazias (marcadas com λ) e outras quatro com os blocos básicos a, b, c e d. Esta organização permite a transferência direta de todas instruções de um bloco para o "buffer" de decodificação: as atividades para alinhar e remover são desnecessárias. Contudo, precisaremos de lógica adicional se desejarmos transferir dois ou mais blocos em paralelo.

3. Ambiente Experimental

BBM é uma máquina hipotética que reconhece o bloco básico como unidade padrão de processamento. Por esta razão, ao invés de um contador de instruções, BBM tem um contador de blocos que é utilizado para buscar o próximo bloco básico. Estes blocos são armazenados em uma memória *cache* de instruções especial. As linhas da *cache* de instruções armazenam um bloco básico integralmente. A modelagem desta *cache* especial e suas vantagens são tópicos abordados em outro artigo.

Após o *fetch* do bloco básico, suas instruções são decodificadas, despachadas, executadas e concluídas. Na máquina BBM, estas tarefas podem ser realizadas seqüencialmente ou em paralelo. Já que queremos investigar o papel desempenhado pelos blocos básicos, a execução seqüencial ou em paralelo destas atividades não é relevante para a apresentação do nosso estudo. Na realidade, a versão BBM que gerou os resultados para este artigo emprega o esquema seqüencial de processamento e sem execução especulativa de blocos básicos. Adicionalmente, somente após a conclusão da última instrução do bloco básico corrente, é que um novo ciclo de máquina é iniciado.

Na realização de nossos experimentos, utilizamos o conjunto de ferramentas "SimpleScalar" [DTBA97]. Conseqüentemente, adotamos o conjunto de instruções suportado pelos

simuladores do SimpleScalar (um superconjunto da arquitetura MIPS-IV) e os utilitários GNU (compilador C e bibliotecas). Os simuladores do SimpleScalar foram modificados para a execução de blocos básicos. Além disto, desenvolvemos uma série de utilitários para identificar os blocos básicos dos programas de teste (codificados em linguagem *Assembly*), para determinar o comprimento de cada bloco básico, para coletar estatísticas de execução e outras tarefas correlatas.

Os resultados apresentados neste artigo foram produzidos com uma versão modificada do simulador "sim-fast" do conjunto SimpleScalar. Esta versão recebe como entrada as estruturas de dados geradas por nossos utilitários, simula o repertório MIPS-IV e produz um perfil da execução. Selecionamos os programas do conjunto SCPEint95 para os experimentos. O conjunto SPECint95 contém três grupos de dados de entrada (teste, treinamento e referência), todos eles utilizados em nossos experimentos. Contudo, se nada for mencionado, os resultados apresentados neste artigo foram obtidos com o conjunto de dados do grupo "referência."

Tabela I: Características do Código Objeto

Programa	Instrução	Bloco Básico
Gcc	272,214	78,580
Compress	12,876	3,499
Go	77,616	14,572
M88ksim	36,128	9,127
Li	22,370	6,156
Perl	66,768	19,208
Ijpeg	49,542	10,665
Vortex	123,692	28,382

Utilizando o simulador e utilitários, determinamos as características estáticas dos programas investigados. A Tabela I lista o total de instruções e blocos básicos de cada programa.

Antes de descrever como os experimentos foram conduzidos, mostraremos um exemplo com algumas figuras de desempenho obtidas em nossas simulações. Este exemplo ilustra como os dados de entrada interferem na utilização dos blocos básicos. A Figura 3 apresenta o percentual de blocos básicos utilizados pela BBM durante a execução dos programas de avaliação.

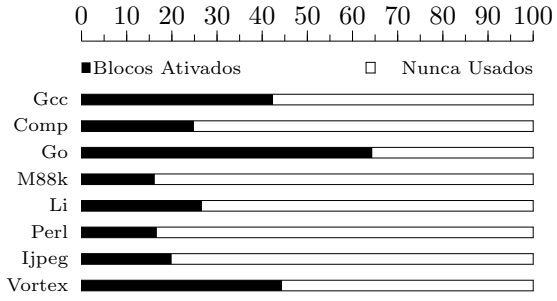

Figura 3: Blocos Básicos Utilizados

Na Figura 3, cada barra representa os blocos básicos do programa de avaliação correspondente. A percentagem de blocos básicos executados (ativados) é representada pela parte escura de cada barra.

Como podemos verificar, exceto para o programa "Go," mais de 50% dos blocos básicos nunca foram utilizados durante a execução. O programa Gcc foi executado pelo simulador BBM com os seguintes parâmetros:
"gcc.ss $flags expr.i -o expr.s", onde:
gcc.ss → programa objeto em código MIPS do compilador C GNU;
$flags → opções de compilação ativadas;
expr.i → texto em C do grupo referência;
expr. s → arquivo saída gerado pelo compilador.

Nesta execução do Gcc, 57,60% dos blocos nunca foram ativados. Em outra execução, com o mesmo arquivo de entrada, mas com as opções de compilação desativadas (com exceção de -O), observamos que 49.002 blocos não foram ativados, ao invés de 45.263 blocos da primeira execução. É oportuno mencionar que nestas duas execuções do Gcc, a diferença entre o total de instruções executadas pela máquina BBM foi muito grande: 1.029 milhões contra 603 milhões de instruções, respectivamente. A explicação para esta diferença é o número de tarefas adicionais que o compilador Gcc necessita realizar conforme especificado pelas opções de compilação.

O grande número de blocos não utilizados junto com a significativa diferença no número de blocos básicos ativados, motivaram este estudo.

4. Experimentos e Resultados

Em todos experimentos, os programas do SPECint95 foram executados integralmente pois a execução parcial poderia fornecer resultados errados no número de blocos não utilizados.

Como visto nos exemplos ilustrados anteriormente, o conjunto de dados de entrada interfere na utilização dos blocos básicos. Ao invés de listar os resultados obtidos com diversos conjuntos de entrada, mostraremos os resultados obtidos com o conjunto de dados especificado na Tabela III.

A seguir, apresentaremos os resultados de cinco conjunto de experimentos. Nesses experimentos utilizamos os programas SPECint95. Dependendo do experimento, uma saída específica foi coletada. Os primeiros dois experimentos avaliam a utilização das instruções e o total de instruções executadas por cada programa de avaliação. Os demais investigam o papel desempenhado pelos blocos básicos.

4.1 Utilização das Instruções

Neste experimento, executamos os programas de avaliação e coletamos as instruções não utilizadas de cada um. A Tabela II lista o total de instruções do código objeto, o número dessas instruções que nunca foram executadas e a percentagem correspondente.

Tabela II: Utilização das Instruções

Programa	Total	Nunca	%
Gcc	272,214	143,181	52.60
Compress	12,876	9,350	72.62
Go	77,616	15,001	19.33
M88ksim	36,128	29,612	81.96
Li	22,370	15,940	71.26
Perl	66,768	54,348	81.40
Ijpeg	49,542	38,774	78.26
Vortex	123,692	62,737	50.72

Podemos ver na Tabela II que na maior parte dos programas, metade das instruções nunca foi utilizada. O programa Go foi a única exceção: mais de 80% de suas instruções foram executadas pelo menos uma vez para um total de 32,7 bilhões (32,7 x 10^9) executadas pelo programa. A percentagem de instruções nunca utilizadas pelos programas Vortex e Gcc (50,7 e 52,6 %) são bastante modestos quando comparados com as outras percentagens.

4.2 Contagem de Instruções

Para este conjunto de experimentos, determinamos o número de instruções executadas por cada programa. Através destes totais é possível determinar o número médio de instruções executadas por cada ativação de bloco básico (os totais de ativações são apresentados na Seção 4.3). A Tabela III apresenta o total de instruções executadas por cada programa.

Tabela III: Características Dinâmicas dos Programas de Avaliação

Programa	Entrada	Contagem
Gcc	cccp.i flags on	1.273.154.334
Compress	bigtest.in	43.064.963.260
Go	50 21 5stone21.in	32.718.644.633
M88ksim	dcrand.big	25.508.326.034
Li	eight queens	956.843.301
Perl	primes.pl primes.in	14.237.931.733
Ijpeg	penguin.ppm	240.717.851
Vortex	vortex.in	9.051.641.850

A Tabela III lista o conjunto dos dados de entrada (do grupo ref.) utilizado em todos experimentos da Seção 4. A última coluna apresenta o total de instruções executado por cada programa. Estes totais justificam o elevado número de horas de processamento requerido pelos programas (alguns deles, exigindo de 2 a 3 dias em uma estação UltraSparc dedicada).

4.3 Ativações dos Blocos Básicos

O próximo conjunto de experimentos está relacionado com o comportamento dos blocos básicos. Quando do início de um ciclo na máquina BBM, a estrutura de dados contendo a descrição dos blocos básicos é examinada. Em seguida, realizamos as atividades que coletam os dados estatísticos.

A determinação do início de um ciclo BBM não é uma tarefa simples: em tempo de execução não é raro descobrir que o próximo bloco básico está ausente da estrutura. Isto significa que nosso utilitário responsável pela construção da estrutura de dados (contendo os blocos básicos do programa estudado) não foi capaz de distinguir, o que parecia ser, um único bloco básico de dois blocos contíguos.

Isto é causado pelos desvios indiretos: neste caso, a última instrução do primeiro bloco básico não é uma transferência de controle e a primeira instrução do segundo bloco é o alvo do desvio indireto. Somente em tempo de execução podemos detectar a existência do segundo bloco básico.

Neste caso, o utilitário é executado novamente, fornecendo-se o endereço do bloco que está faltando. Após o reparo da estrutura, a simulação é iniciada novamente.

No conjunto subseqüente de experimentos, determinamos quantas vezes os blocos básicos foram ativados. A Tabela IV lista o total destas ativações.

Tabela IV: Total de Ativações dos B. Básicos

Programa	Blocos	Usados	Ativações
Gcc	78.580	34.206	285.850.621
Compress	3.499	902	10.435.297.613
Go	14.572	10.729	5.500.462.332
M88ksim	9.127	1.480	4.347.669.950
Li	6.156	1.641	250.115.506
Perl	19.208	3.203	3.173.078.887
Ijpeg	10.665	2.128	33.336.645
Vortex	28.382	12.600	1.722.935.280

As colunas da Tabela IV listam o total de blocos de cada programa, quantos deles foram ativados pelo menos uma vez e o número de ciclos BBM para executá-los.

Por exemplo, o programa Perl é formado por 19.208 blocos mas apenas 3.203 destes foram ativados, e o total de vezes que ocorreu uma transferência de controle para um deles foi 3,173 bilhões.

4.4 Blocos Básicos Mais Usados

O número total de ativações de cada programa não é uniforme. No próximo conjunto de experimentos selecionamos os blocos mais freqüentemente ativados.

A Tabela V mostra o número de blocos básicos cuja soma de ativações alcançou 90% do total de ativações.

Por exemplo, na linha relacionada ao programa Ijpeg, pode-se ver que ele é formado por 10.665 blocos e que, em tempo de execução, apenas 2.128 destes foram ativados e 41 são responsáveis por 90% de todas as ativações de blocos básicos.

Tabela V: Blocos Básicos Responsáveis por 90% das Ativações

Programa	Blocos	Usados	90%
Gcc	78.580	34.206	4.952
Compress	3.499	902	43
Go	14.572	10.729	1.093
M88ksim	9.127	1.480	144
Li	6.156	1.641	140
Perl	19.208	3.203	117
Ijpeg	10.665	2.128	41
Vortex	28.382	12.600	483

A Figura 4 mostra a percentagem de blocos básicos que foram ativados pelo menos uma vez, junto com o percentual de blocos básicos responsáveis por 90% das ativações.

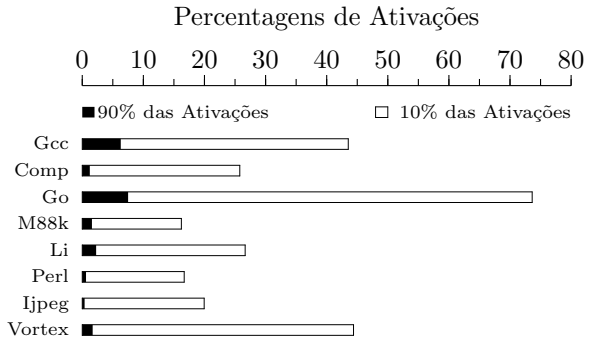

Figura 4: Blocos Responsáveis pelas Ativações

Podemos constatar na Figura 4 que a percentagem de blocos responsáveis por 90% das ativações (a parte escura das barras) é muito modesta. Elas variam de 0,38 até 2,27% para seis dos programas avaliados. Os programas Gcc (6,30%) e Go (7,50%) foram as exceções. Mesmo assim, estas percentagens são muito menores que aquelas apresentadas em [JHDP90]. Estes resultados, originalmente aqui divulgados, são bastante promissores. Eles explicam por exemplo, as elevadas taxas de acerto em *caches* de instruções (e também pelas técnicas de previsão de desvios).

4.5 Tamanho dos Blocos Básicos

Na seção anterior verificamos que uma pequena fração de blocos básicos é responsável por 90% das ativações. Contudo, poderíamos argumentar que estes blocos básicos são formados por um grande número de instruções.

Este é o tópico abordado por um outro artigo, mas se examinarmos os tamanhos dos blocos básicos mais freqüentemente ativados, poderemos ter uma melhor visão do que realmente acontece.

A Tabela VI lista o número total de ativações de cada um dos programas de teste em conjunto com a soma das ativações dos blocos básicos contendo 1, 2, 3 e 4 instruções e que foram responsáveis por 90% das ativações.

Os valores na Tabela VI estão representados em milhões de instruções. Podemos observar que os blocos mais executados contêm 1 ou 2 instruções. O M88ksim é a única exceção: blocos básicos com 4 instruções são os mais freqüentemente executados.

5. Conclusões

Os principais conceitos de BBM –uma arquitetura especializada na execução de blocos básicos– foram apresentados.

Experimentos envolvendo a execução do conjunto de avaliação SPECint95 na BBM revelaram que um número significativo de instruções permaneceu sem utilização durante a execução.

O número de instruções não utilizadas depende do conjunto de entrada. Usualmente esse número representa mais de 50% do código objeto e, para a maioria dos programas pesquisados, o número de instruções sem uso excedeu 70% do código.

Comportamento análogo é verificado também para os blocos básicos destes programas. Além disto, nossos experimentos mostraram que na maioria dos programas de avaliação, menos que 3% dos blocos básicos são responsáveis por mais de 90% das ativações.

Esses resultados sugerem que estruturas de *hardware* que armazenam as instruções sejam organizadas de modo que cada bloco básico fique isolado dos outros. As unidades de busca e os mecanismos de predição poderiam se beneficiar desta organização.

O conceito explorado por uma arquitetura do tipo BBM é um tópico de investigação muito promissor, e há muitos caminhos de pesquisa a serem trilhados.

Tabela VI: Ativações de Blocos Básicos segundo o Tamanho (em milhões)

Programa	Total	1 Instrução	2 Instruções	3 Instruções	4 Instruções
Gcc	285,850	39,733	72,194	47,096	36,537
Compress	10.435,297	2.428,788	2.307,591	825,180	1.772,252
Go	5.500,462	1.027,641	675,693	437,255	625,886
M88ksim	4.347,669	330,113	844,877	301,066	1.033,122
Li	250,115	35,883	72,143	52,697	27,025
Perl	3.173,078	348,666	548,894	532,609	547,504
Ijpeg	33,336	2,273	9,188	2,900	6,197
Vortex	1.722,935	374,319	463,707	156,664	73,782

Bibliografia

[CMMP95] Thomas M. Conte, Kishore N. Menezes, Patrick M. Mills, Burzin A. Patel, "Optimization of Instruction Fetch Mechanisms for High Issue Rates," Proceedings of the 22nd Annual International Symposium on Computer Architecture, ISCA '95, Santa Margherita Ligure, Italy, June 1995, pp. 333–344.

[DENN70] Peter J. Denning, "Virtual Memory," Computing Surveys, Vol.2, September 1970, pp. 153–189.

[DBTA97] Doug Burger and Todd M. Austin, "The SimpleScalar Tool Set, Version 2.0," Technical Report #1342, Computer Sciences Department, University of Wisconsin-Madison, June 1997, pp. 1–21.

[ERBS96] Eric Rotenberg, Steve Bennet, and James E. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," Proceedings of the 29th Annual International Symposium on Microarchitecture, Paris, France, December 1996, pp. 24–34.

[FISH84] Joseph A. Fisher, "VLIW Machine: A Multiprocessor for Compiling Scientific Code," Computer, July 1984, pp. 45–53.

[JHDP90] John L. Hennessy and David A. Patterson, "Computer Architecture: a Quantitative Approach," Morgan Kaufmann Publishers, Inc., San Mateo, California, U.S.A., 1st Edition, 1990.

[KNUT71] D. E. Knuth, "An Empirical Study of FORTRAN Programs," Software—Practice & Experience, Vol.1, 1971, pp. 105–133.

[LDSM80] David Landskov, Scott Davidson, Bruce Shriver, and Patrick W. Mallet, "Local Microcode Compaction Techniques," Computing Surveys, Vol. 12, No. 3, September 1980, pp. 261–294.

[SJSM96] André Seznec, Stéphan Jourdan, Pascal Sainrat, and Pierre Michaud, "Multiple-Block Ahead Branch Predictors," Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems, ASPLOS, 1996, pp. 116–127.

[SMSP88] Stephen W. Melvin, Michael C. Shebanov, and Yale N. Patt, "Hardware Support for Large Atomic Units in Dynamically Scheduled Machines," Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecure, San Diego, CA., November 1988, pp. 60–63.

[SPEP98] Sanjay J. Patel, Marius Evers, and Yale N. Patt, "Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing," Proceedings of the 15th International Symposium on Computer Architecture, Barcelona, Spain, June 1988.

[SPFP97] Sanjay J. Patel, Daniel H. Friendly, and Yale N. Patt, "Critical Issues Regarding the Trace Cache Fetch Mechanism," Technical Report CSE-TR-335-97, University of Michigan, May 1997, pp. 1–33.