Reprodução de um elemento do artigo *Accelerating Decoupled Look-ahead via Weak Dependence Removal: A Metaheuristic Approach*

Gustavo Ciotto Pinton

Universidade Estadual de Campinas - UNICAMP
MO601B - Arquitetura de Computadores

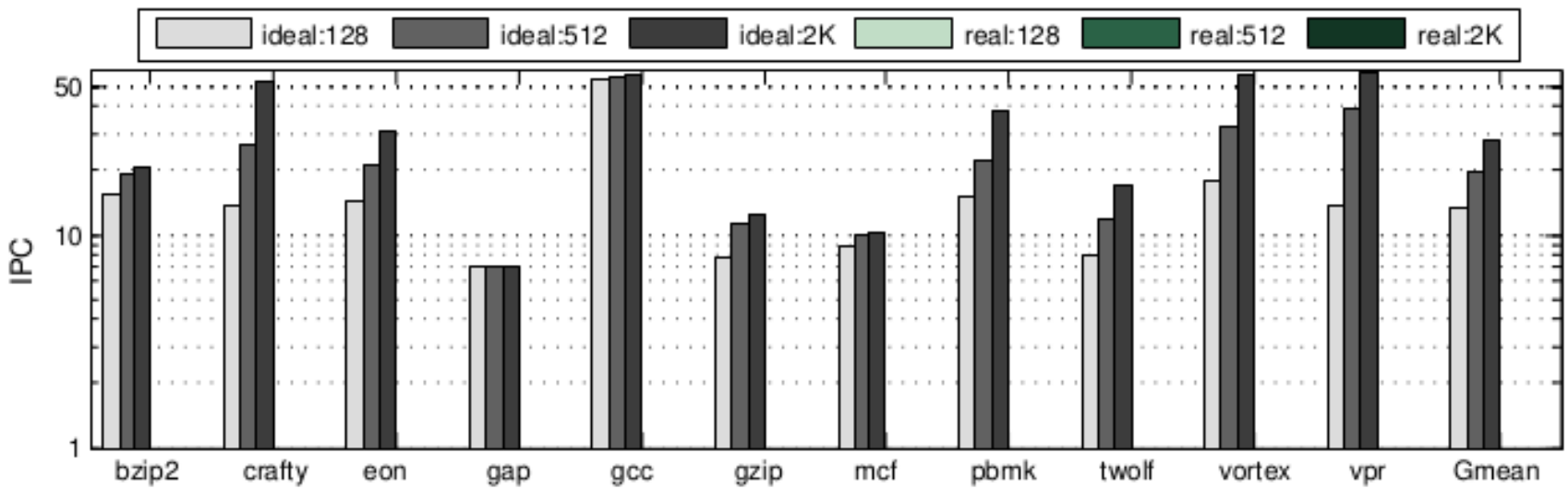
18 de Novembro de 2016

Sumário

Outline

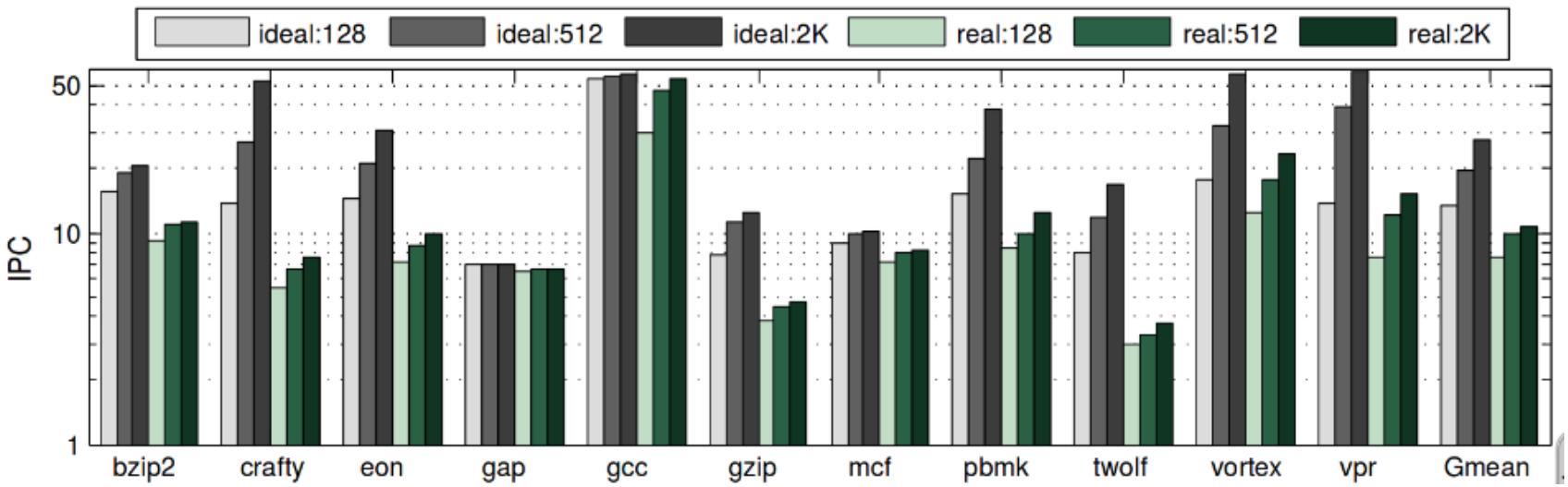
Motivação

- Apesar da proliferação de sistemas *multi-core* e *multi-threaded*, a performance de aplicações *single-thread* ainda é um objetivo importante.
- Aplicações *single-threaded* fazem uso do paralelismo de instruções.
- Desafios: como explorar este paralelismo sem muitos custos adicionais
- Alternativa: *Decoupled look-ahead architecture*



Motivação

- Objetivos da arquitetura *decoupled look-ahed thread*.
  - Minimizar os custos de *branch mispredictions* e *cache misses*, por exemplo.
  - Explorar oportunidades de paralelismo e o grau de dependência de instruções.
  - Minimizar o consumo energético



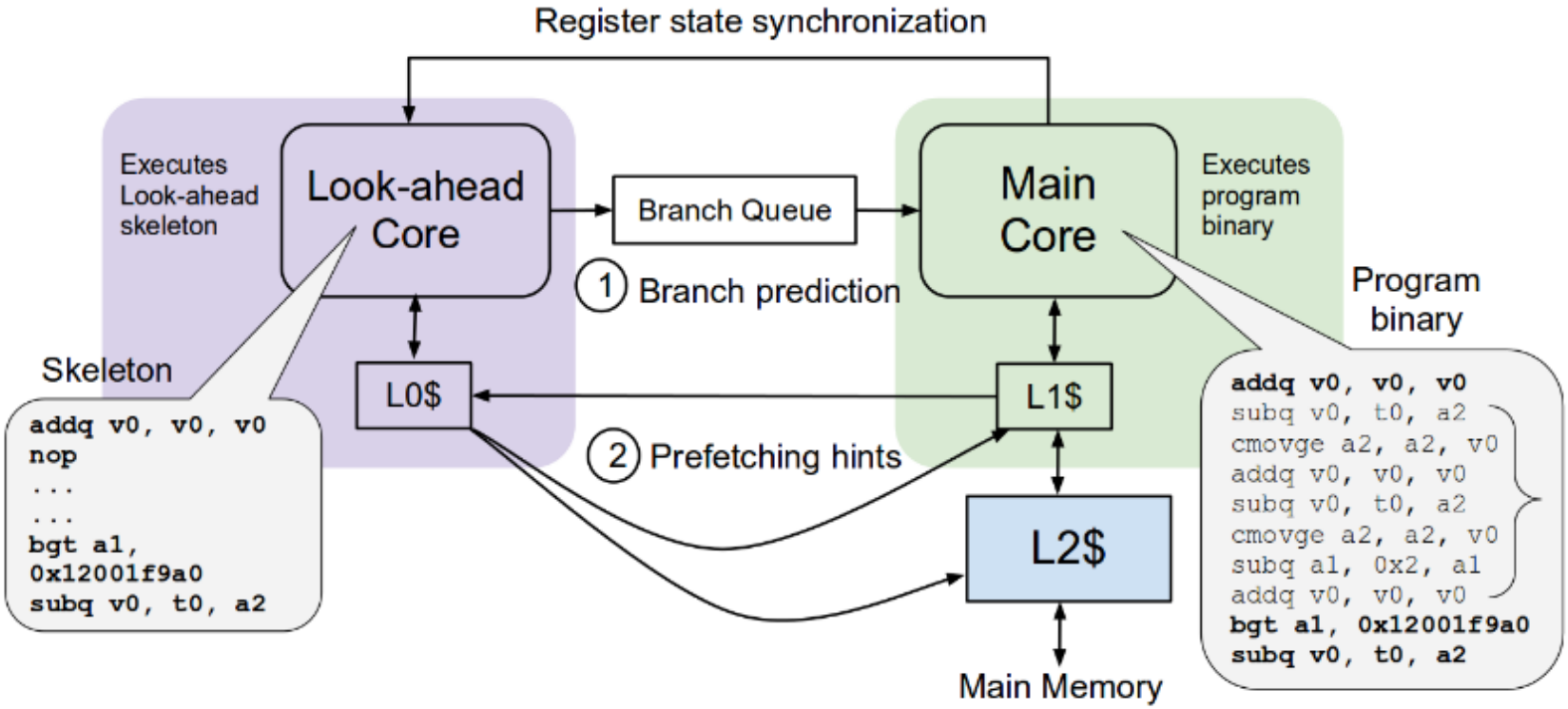
Motivação

- Constatou-se que a *thread* auxiliar, isto é, a *look ahead thread* se tornou o novo limite de velocidade do sistema.
- A corretude da *look-ahead thread* não é exigida, permitindo várias otimizações
  - Dependências fracas: instruções que contribuem marginalmente para o resultado e, portanto, podem ser retiradas.
- O artigo *Accelerating Decoupled Look-ahead via Weak Dependence Removal: A Metaheuristic Approach* propõe uma maneira de otimizá-la:
  - Identificação dos pontos desnecessários que poderiam ser retirados desta *thread*.
  - Como identificá-los automaticamente?
  - Uso de algoritmos genéticos.
  - Como caracterizar um gene?

Outline

Arquitetura *Decoupled Look-Ahead*

- *Parser* que transforma o binário do programa principal em uma versão reduzida, somente para procurar *misses*.
- Versão esqueleto roda em *core* separado, anteriormente ao programa principal.
- Os resultados de saltos condicionais são transmitidos através de uma fila para o *core*.



. Garg and M. Huang, "A Performance-Correctness Explicitly Decoupled Architecture", *MICRO-08*

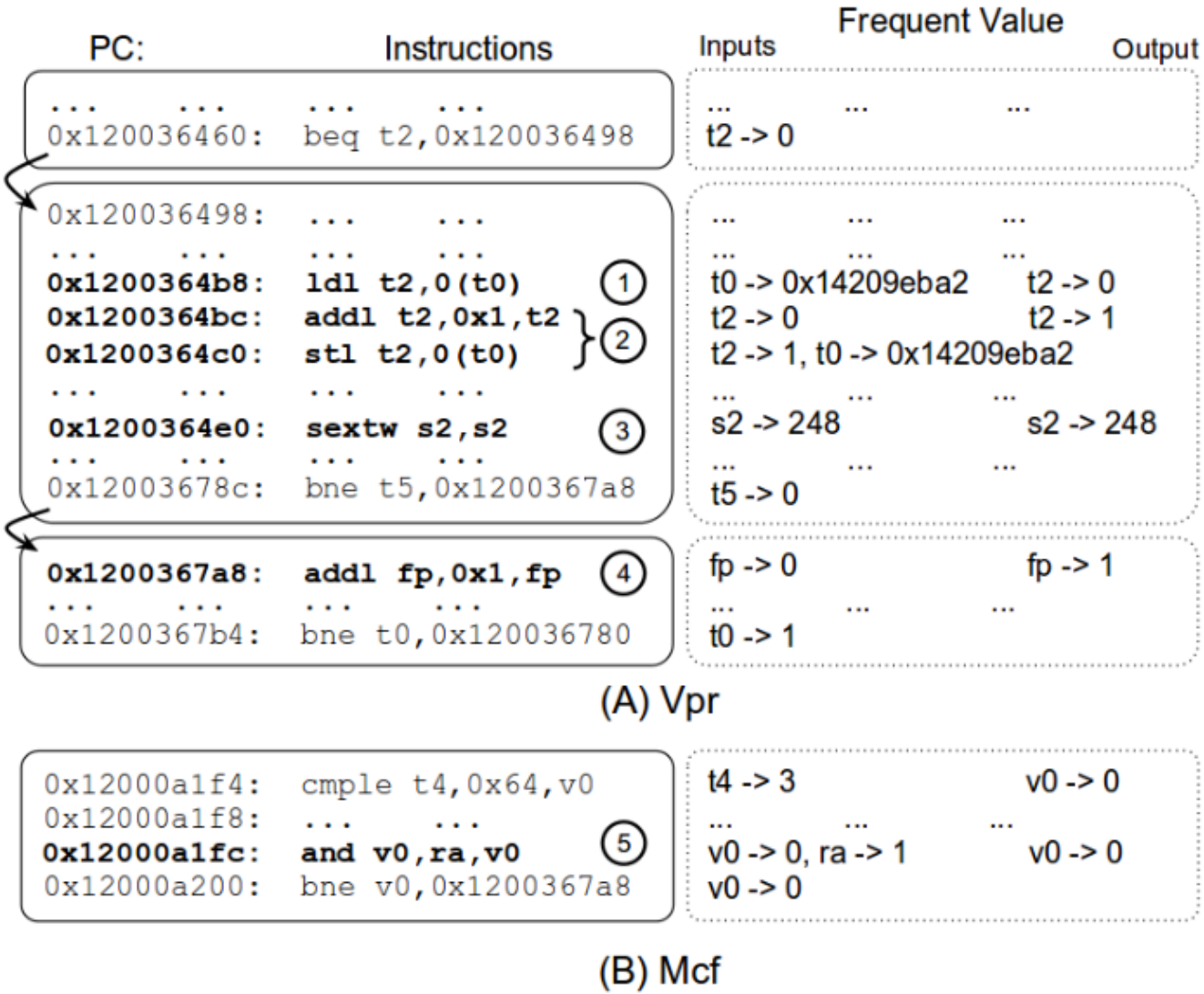
Arquitetura *Decoupled Look-Ahead*

Dependências fracas

- Instruções que contribuem o mínimo para os propósitos da *lookup ahead thread*.
- Exemplos:
  - Instruções aritméticas e lógicas que não mudam o resultado de um registrador na maior parte do tempo: casos 3 e 5.
  - Ajustes inúteis em registradores (realizar uma operação em um registrador sendo que ele será o alvo de um store em seguida): casos 2 e 4.
  - *Loads/Stores* inúteis: caso 1.

Arquitetura *Decoupled Look-Ahead*

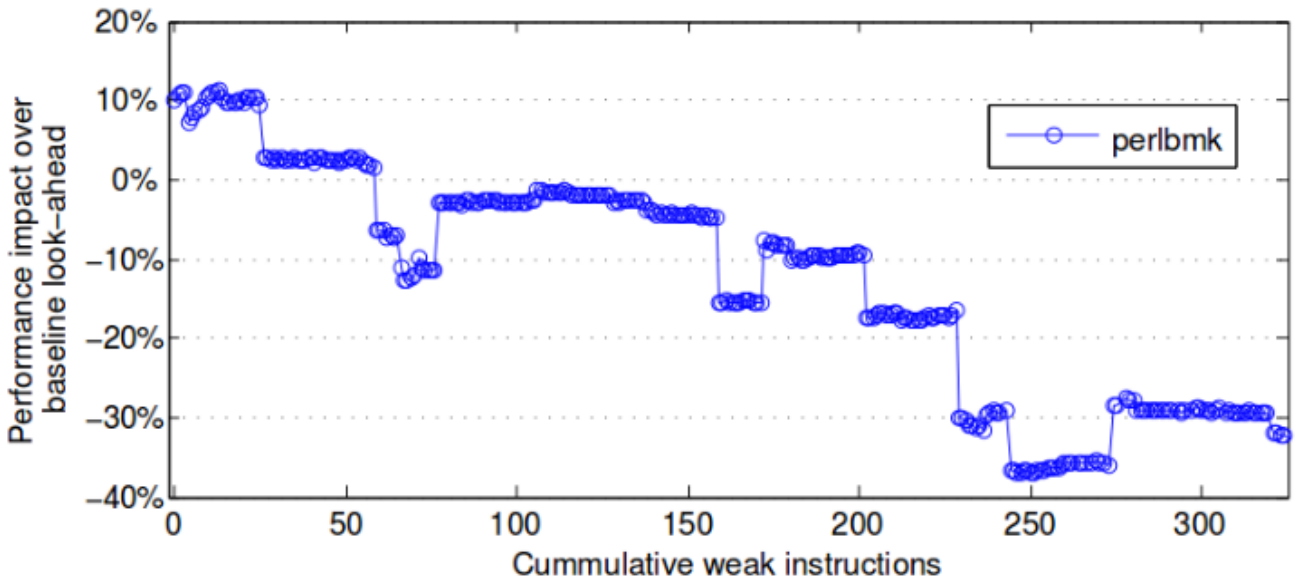
Dependências fracas



Arquitetura *Decoupled Look-Ahead*

Dependências fracas

- Análise muito difícil caso seja realizada estaticamente: uma instrução se torna uma dependência fraca de acordo com o contexto do programa.
- Nenhuma característica especial em comum que pudesse identificá-las no momento de geração dos esqueletos.
- A identificação de uma instrução errada pode piorar o desempenho do sistema, dado que novos *misses* podem ser inseridos.



Arquitetura *Decoupled Look-Ahead*

Dependências fracas

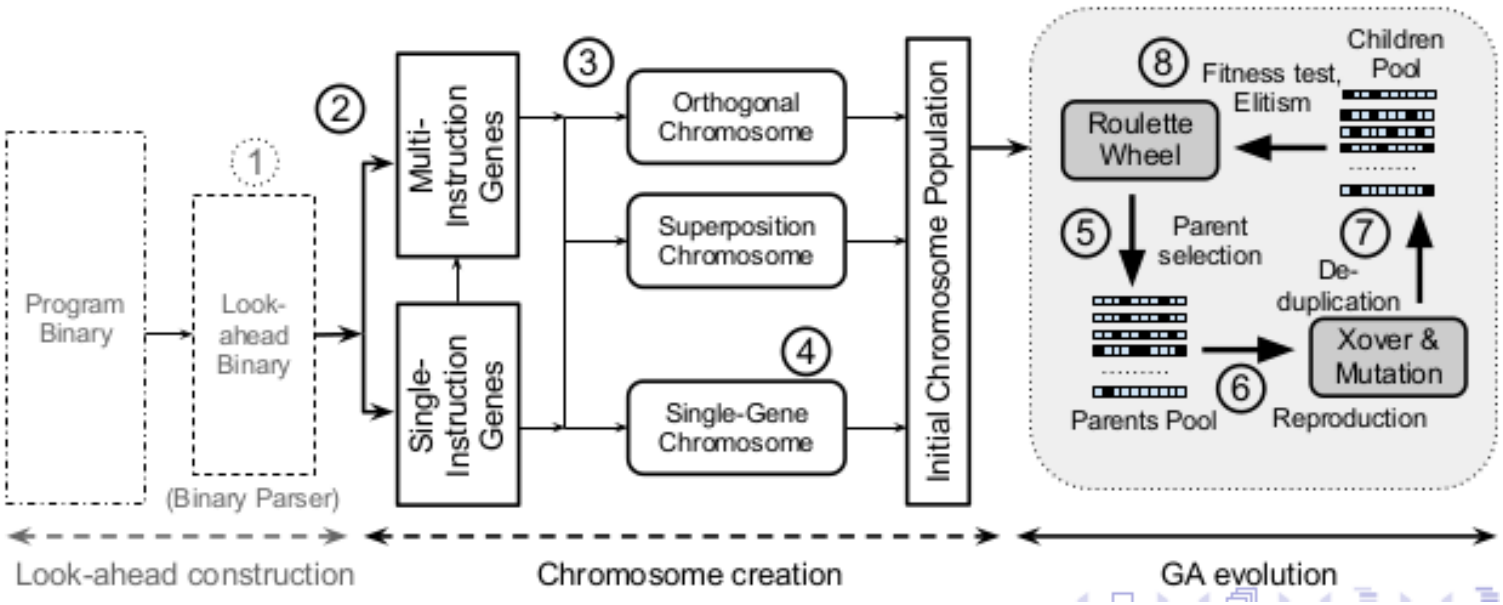
- Na figura anterior:
  - Após a remoção de 50 dependências fracas, o efeito torna-se negativo.
  - Em torno de 250 instruções removidas, o pior efeito é encontrado (40% de degradação)
- A identificação de um falso positivo é extremamente custoso!
- Dada à natureza dinâmica da evolução das dependências, uma boa maneira de encontrar as dependências corretas é o uso de algoritmos genéticos.

Outline

Implementação Proposta

Design básico

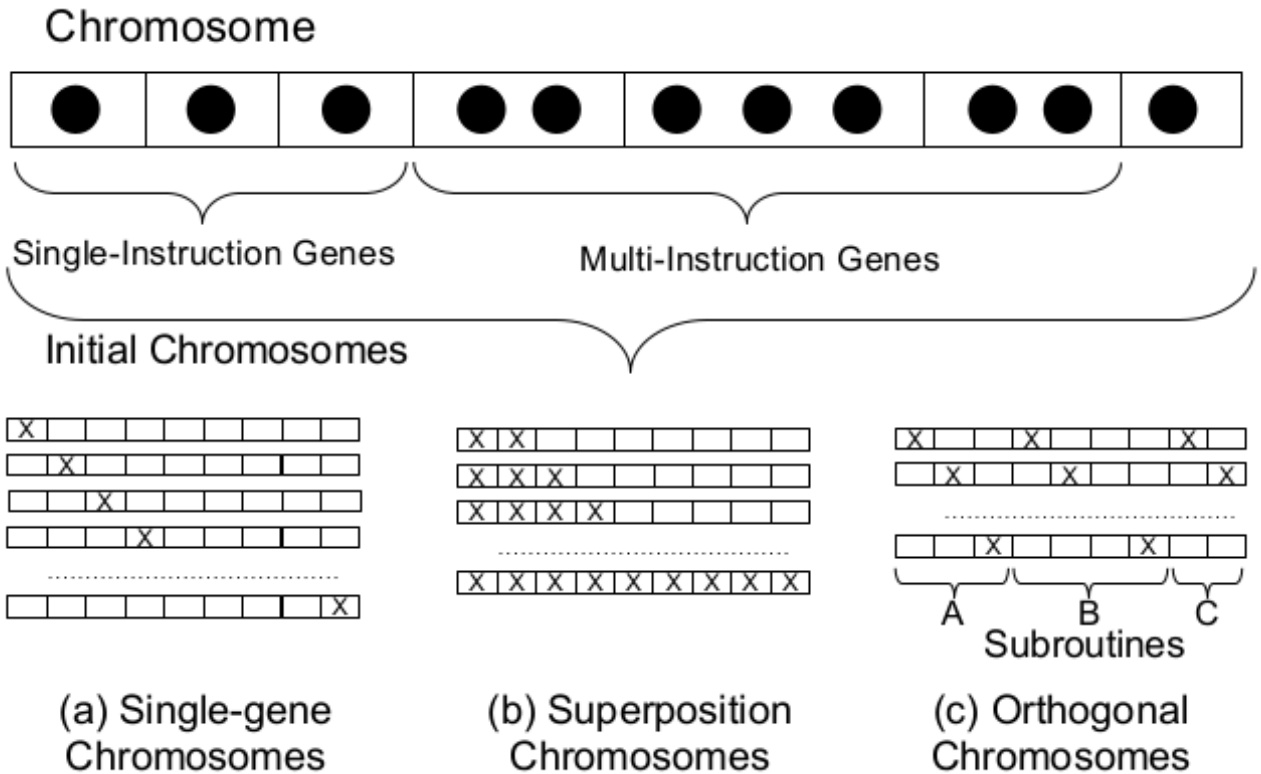
- Objetivo: encontrar o esqueleto que maximiza a performance.
- Esqueleto = Versão do programa principal após um *filtro*
- Algoritmo genético para encontrar o melhor *filtro*:
  - Esqueleto é representado por um vetor de *bits*
  - Mapeamento: instrução fraca $\rightarrow$ gene, coleção de instr. $\rightarrow$ cromossomo



Implementação Proposta

Design básico

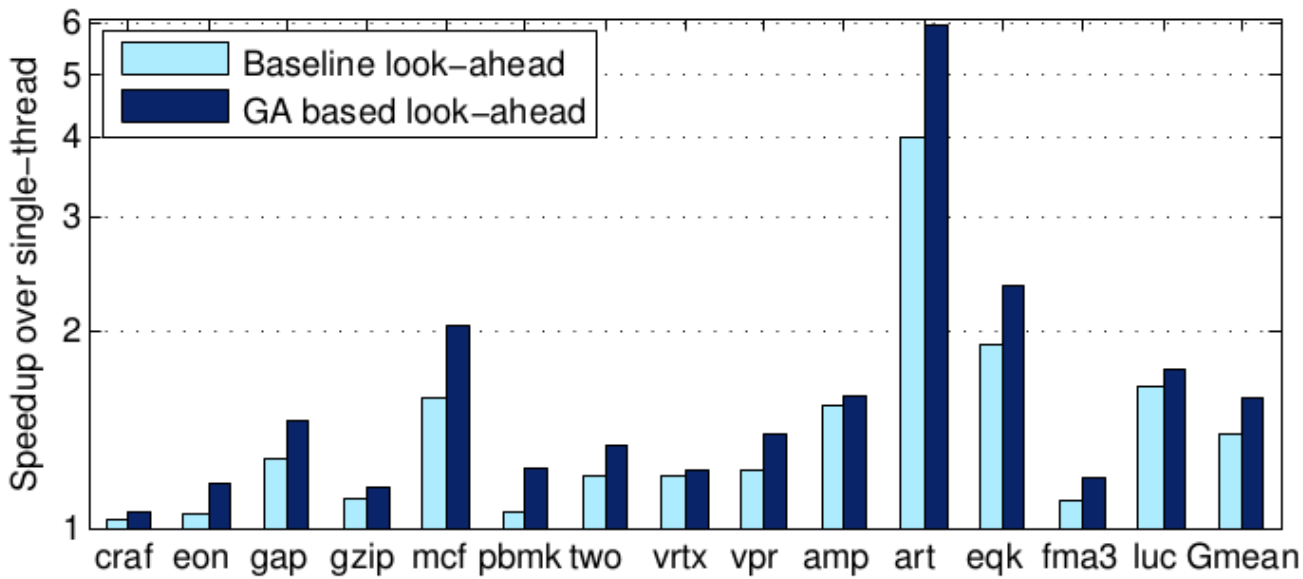
- Formação da população inicial:
  - *Single-Gene chromosomes*: N cromossomos com apenas um gene
  - *Superposition chromosomes*: N - 1 cromossomos gerados a partir da superposição dos genes em ordem crescente de *fitness*
  - *Orthogonal chromosomes*: genes obtidos de rotinas distintas



Outline

Resultados

- Aumento do desempenho sobre os programas originais: 1.58x
- Aumento do desempenho sobre o modelo de *decoupled look-ahed thread* original: 1.14x (em média geométrica)



Outline

Reprodução de um elemento do artigo

Apresentação

- Reprodução das curvas *ideal* e *single-thread* da figura 3.

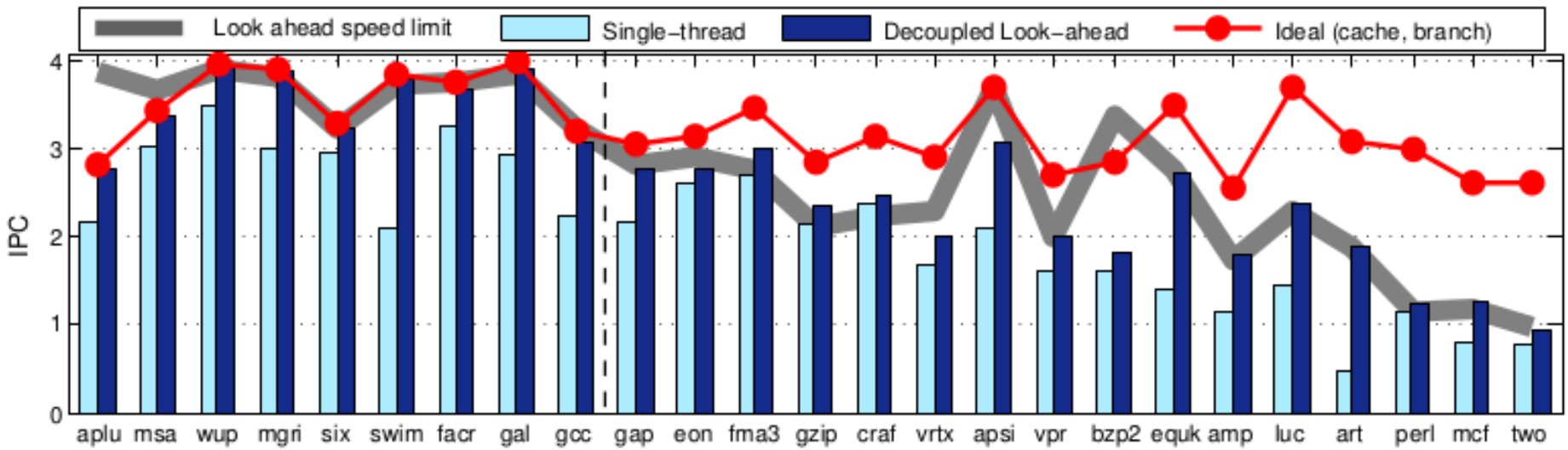


Figure 3. Performance comparison of 4 configurations. Shown in the bars are baseline single core (left) and a decoupled look-ahead system (right). Two upper-bounds are shown: the performance of a single core with idealized branch predictions and perfect cache accesses (curve with circles), and the the approximate speed limit of the look-ahead thread (gray wide curve indicating approximation). The applications are sorted with increasing performance gap between the decoupled look-ahead system and the prediction- and accesses-idealized single-core system.

Outline

Reprodução de um elemento do artigo

Configuração do *sniper*

- Máquina virtual de 64 bits Ubuntu 14.04 LTS.
- Compilado com o gcc 4.4.
- Dois tipos de pinballs do `SPEC CPU2006` foram utilizados:
  - 100M instruções de warmup e 30M instruções na região detalhada.
  - 1B de instruções na região detalhada e sem warmup.
- Integração dos *pinballs* com o *sniper*:
  - *Pinballs* de 1B de instruções: apenas acrescentar `-pinballs=<...>`
  - *Pinballs* de 100M+30M de instruções: diversas tentativas

Reprodução de um elemento do artigo

Pinballs de 100M+30M

- Opção `-roi`: o *pinball* é executado integralmente em modo `CACHE_ONLY`, gerando nenhum arquivo de saída e nenhuma estatística. ✗
- Implementação de *pintool*: conta as instruções e chama a função `SimRoiStart()` explicitamente (`sniper/include/sim_api.h`) ✗
  - *Sniper* não é capaz de executar o binário do pin, uma vez que este último foi pré-compilado em uma máquina de 32 bits.
- Máquina virtual de 32 bits: compilação do *sniper*. ✗
  - *Pinballs* criados em máquinas de 64 bits e não podem rodar em máquinas de 32 bits.
- *Script* python com a opção `-s`: script python capaz de contar as instruções é disponível no diretório `sniper/scripts`. ✓
  - `roi-icount.py`: deve ser chamado com o parâmetro 0:100000000:30000000

Reprodução de um elemento do artigo

Ambiente de testes

- Linhas indicadas por ✓: corretamente configuradas *sniper*.
- Linhas indicadas por ✗: modificações mais profundas no modelo de intervalos do *sniper*.

Baseline core		
Fetch/Decode/Issue/Commit	8/4/6/6	✗
ROB	128	✓
Functional units	INT 2+1 mul +1 div, FP 2+1 mul +1 div	✗
Fetch Q/ Issue Q / Reg. (int,fp)	(32, 32) / (32, 32) / (80, 80)	✗
LSQ(LQ,SQ)	64 (32,32) 2 search ports	✗
Branch predictor	Gshare – 8K entries, 13 bit history	✓
Br. mispred. penalty	at least 7 cycles	✓
L1 data cache (private)	32KB, 4-way, 64B line, 2 cycles, 2 ports	✓
L1 inst cache (private)	64KB, 2-way, 128B, 2 cycles	✓
L2 cache (shared)	1MB, 8-way, 128B, 15 cycles	✓
Memory access latency	200 cycles	✓

Reprodução de um elemento do artigo

Arquivos de configuração

- *Core* modelado pela microarquitetura *Nehalem* (padrão do *sniper*)
  - Modelar POWER5 : projeto 4!
- *Caches* ideais:
  - `perf_model/tlb/penalty = 0`
  - `perf_model/l1_icache/perfect=true`
  - `perf_model/l1_dcache/perfect=true`
  - `perf_model/l2_cache/perfect=true`
- *Branch predictor* ideal:
  - `perf_model/branch_predictor/mispredict_penalty=0`

Reprodução de um elemento do artigo

Implementação do preditor *gshare*

- Definição da classe abstrata `BranchPredictor`, possuindo dois métodos virtuais:
  - `predict()` : retorna um *booleano*
  - `update()`: atualiza preditor de acordo com o resultado.
- Implementação disponível: preditor de dois *bits* `SaturatingPredictor<2>`
- `predict()`:
  - Cálculo do índice: `indice = (pc XOR global_history) & 0x1FFF`
  - Retornar `preditores[indice].predict()`
- `update()`:
  - Cálculo do índice, conforme acima.
  - Atualização do registro global:
    `global_history = (global_history « 1) | branch_outcome`
  - Atualizar preditor: `preditores[indice].update(branch_outcome)`
- Integração com o *sniper*: através do arquivo `branch_predictor.cc`.

Outline

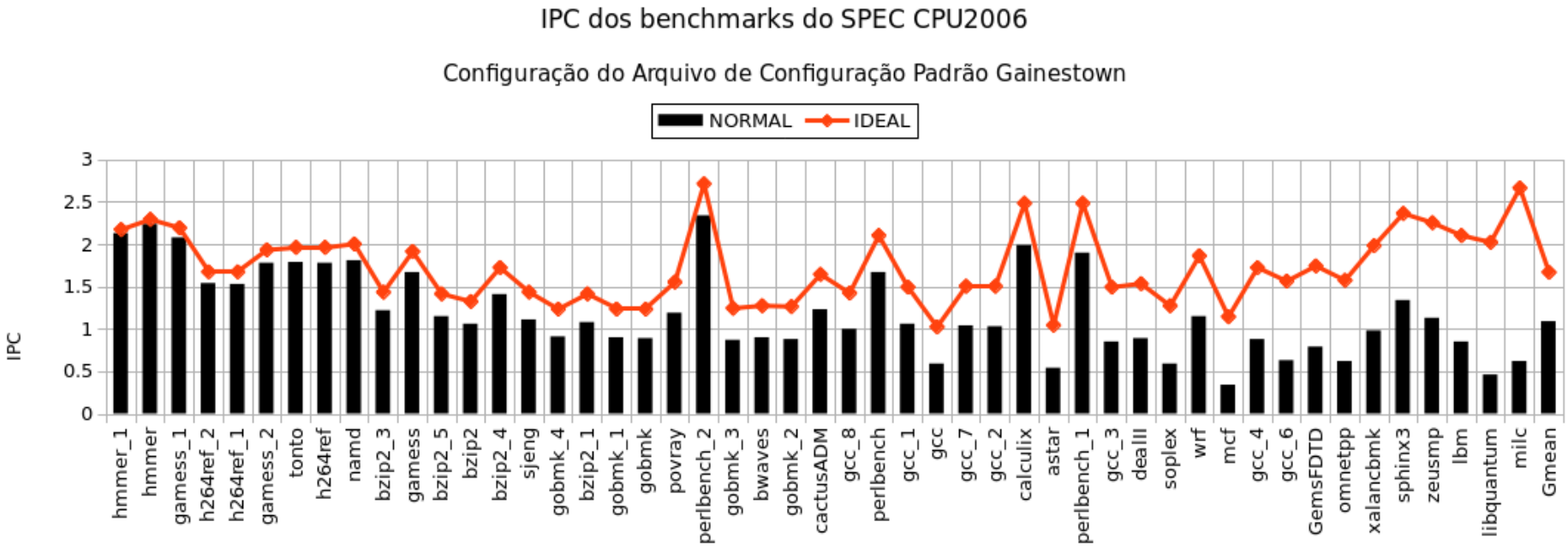
Reprodução de um elemento do artigo

Resultados

- Uso dos *benchmarks* do `SPEC CPU2006`, contrariamente ao artigo.
- Apesar de todo trabalho para configurar os *pinballs* de 100M+30M, eles não foram utilizados
  - Resultados muito distantes daqueles encontrados para os de 1B e para os do artigo.
- Duas configurações utilizadas:
  - Microarquitetura *Intel Gainestown* para testes
  - Configuração do artigo
- *Pinballs* de 1B: média de 30 minutos para cada entrada
  - Execução em 4 *cores*

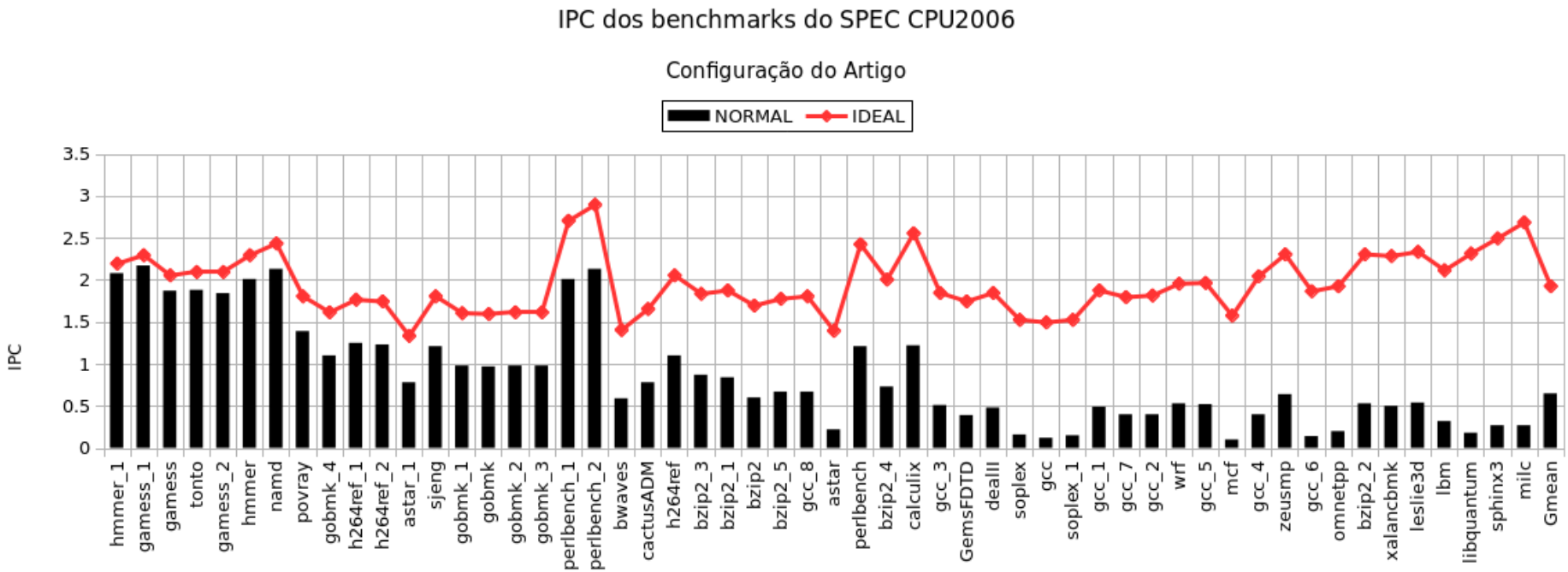
Reprodução de um elemento do artigo

Resultados - *Gainestown*



Reprodução de um elemento do artigo

Resultados - *Artigo*



Outline

Reprodução de um elemento do artigo

Conclusão

- Desempenhos ideais: IPCs muito parecidos entre si.
  - Artigo: 1.93 (*Gmean*).
  - *Gainestown*: 1.68 (*Gmean*).
- As medidas de IPC para os casos normais variam mais entre si:
  - Artigo: 0.65 (*Gmean*).
  - *Gainestown*: 1.09 (*Gmean*).
  - Variação de 1.68x.
- Em relação aos resultados encontrados no artigo:
  - IPCs maiores que os encontrados nos experimentos encontrados neste relatório.
  - *Gainestown*: 1.09 (*Gmean*).
  - Autores utilizaram os programas completos!
- No nosso caso, uma entrada completa levaria $\approx$ 500 horas!

Referências

- Carison, T. E. (2012). Interval simulation. `http://snipersim.org/w/Interval_Simulation`.
- Carison, T. E. and Heirman, W. (2013). The Sniper User Manual.
- Parihar, R. and Huang, M. C. (2014). Accelerating decoupled look-ahead via weak dependence removal: A metaheuristic approach. International Symposium on High Performance Computer Architecture.