UFMG

40 ANOS 1976-2016
UFMG | ICEx
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM
CIÊNCIA DA COMPUTAÇÃO

Programming Language Laboratory

Interdependência entre Alocação de Registradores e Escalonamento de Instruções: Estudo Sistemático e Verificação de Soluções

João Francisco Neiva de Carvalho

Orientadora: Profª. Mariza Andrade da Silva Bigonha

1. Introdução
2. Revisão Sistemática da Literatura
3. Solução escolhida
4. Implementação
5. Testes e análise dos resultados
6. Conclusão

Introdução

Duas tarefas na geração de código:

Alocação de registradores: mapeamento dos pseudoregistradores do código intermediário para os registradores físicos da máquina.

Escalonamento de instruções: ordenação das instruções do código para aproveitar paralelismos e minimizar o tempo de execução do programa.

Se o escalonamento de instruções é feito antes da alocação de registradores (método *prepass*), o tempo de permanência das variáveis nos registradores pode ser desnecessariamente longo e, em função dessa excessiva ocupação, o número de registradores físicos e o número de acessos à memória podem aumentar.

Por outro lado, se a alocação de registradores é realizada primeiramente (método *postpass*), variáveis de instruções independentes, que poderiam ser executadas em paralelo, podem ser atribuídas a um mesmo registrador, criando falsas dependências entre as instruções e, assim, limitando as possibilidades de escalonamento.

```
f = a * b
g = (c + d) * (a + e)
h = f + g
```

Código fonte

```
1  Load PR1, a
2  Load PR2, b
3  Mul  PR3, PR1, PR2
4  Load PR4, c
5  Load PR5, d
6  Add  PR6, PR4, PR5
7  Load PR7, e
8  Add  PR8, PR1, PR7
9  Mul  PR9, PR6, PR8
10 Add  PR10, PR3, PR9
11 Stor PR10, h
```

PR: Pseudoregistrador

Código intermediário

Extraído de:
James R. Goodman, Wei-Chung Hsu
Code Scheduling and Register Allocation in Large Basic Blocks, 1988

Método *Prepass*

```
1  Load PR1, a
2  Load PR2, b
3  Mul  PR3, PR1, PR2
4  Load PR4,c
5  Load PR5, d
6  Add  PR6, PR4, PR5
7  Load PR7, e
8  Add  PR8, PR1, PR7
9  Mul  PR9, PR6, PR8
10 Add  PR10, PR3, PR9
11 Stor PR10, h
```

Código intermediário

```
4  Load PR4, c
5  Load PR5, d
7  Load PR7, e
1  Load PR1, a
2  Load PR2, b
6  Add  PR6, PR4, PR5
8  Add  PR8, PR1, PR7
3  Mul  PR3, PR1, PR2
9  Mul  PR9, PR6, PR8
10 Add  PR10, PR3, PR9
11 Stor PR10, h
```

Código após o escalonameto

```
4  Load R1, c
5  Load R2, d
7  Load R3, c
1  Load R4, a
2  Load R5, b
6  Add  R1, R1, R2
8  Add  R3, R4, R3
3  Mul  R4, R4, R5
9  Mul  R1, R1, R3
10 Add  R4, R4, R1
11 Stor R4, h
```

Código final, após a alocação

Método *Postpass*

```
1  Load PR1, a
2  Load PR2, b
3  Mul  PR3, PR1, PR2
4  Load PR4,c
5  Load PR5, d
6  Add  PR6, PR4, PR5
7  Load PR7, e
8  Add  PR8, PR1, PR7
9  Mul  PR9, PR6, PR8
10 Add  PR10, PR3, PR9
11 Stor PR10, h
```

Código intermediário

```
1  Load R1, a
2  Load R2, b
3  Mul  R2, R1, R2
4  Load R3, c
5  Load R4, d
6  Add  R3, R3, R4
7  Load R4, e
8  Add  R1, R1, R4
9  Mul  R1, R1, R3
10 Add  R1, R1, R2
11 Stor R1, h
```

Código após a alocação

```
4  Load R3, c
5  Load R4, d
1  Load R1, a
2  Load R2, b
6  Add  R3, R3, R4
3  Mul  R2, R1, R2
7  Load R4, e
8  Add  R1, R1, R4
9  Mul  R1, R1, R3
10 Add  R1, R1, R2
11 Stor R1, h
```

Código final, após o escalonamento

1. Compreender o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções, avaliando seus efeitos sobre a otimização do código e identificando na literatura os principais métodos e abordagens para lidar com ele.

2. Dentre os métodos encontrados, identificar o que melhor soluciona o problema e implementá-lo em um compilador real, verificando a sua eficiência por meio de testes e comparações com os outros métodos existentes no compilador.

Revisão Sistemática da Literatura

Revisão sistemática de literatura: “É uma forma de identificar, avaliar e interpretar todas as pesquisas disponíveis que são relevantes a uma particular questão de pesquisa, área, ou fenômeno de interesse”.

Barbara Ann Kitchenham
Technical Report TR/SE-0401, 2004

Estudos primários: investigam diretamente uma questão de pesquisa. São os estudos que contribuem para a revisão sistemática.

Estudos secundários: revisam os estudos primários relacionados a um determinado tema de pesquisa, visando integrar, ou sintetizar as evidências relatadas. Uma revisão sistemática da literatura é uma forma de estudo secundário.

1. Utiliza um *protocolo de revisão*, que nada mais é do que um plano que descreve como a revisão sistemática será conduzida. O protocolo de revisão define:
   - A estratégia de busca dos estudos primários, isto é, a *string* de busca a ser utilizada e as bases de dados a serem pesquisadas.
   - Os critérios de inclusão e exclusão para a avaliação e seleção dos estudos primários.
   - As questões de pesquisa que devem ser respondidas com a análise dos estudos primários selecionados.
   - As informações que deverão ser extraídas de cada estudo primário selecionado, incluindo os critérios qualitativos pelos quais tais estudos serão avaliados.
2. É um pré-requisito para meta-análises quantitativas.

Planejamento: definição do protocolo de revisão.

Execução: busca e seleção dos estudos primários utilizando o protocolo de revisão definido.

Análise: extração dos dados dos estudos selecionados e resposta das questões de pesquisa.

Processo de seleção realizado em quatro etapas:

Etapa 1: identificação de estudos duplicados. Etapa automatizada.

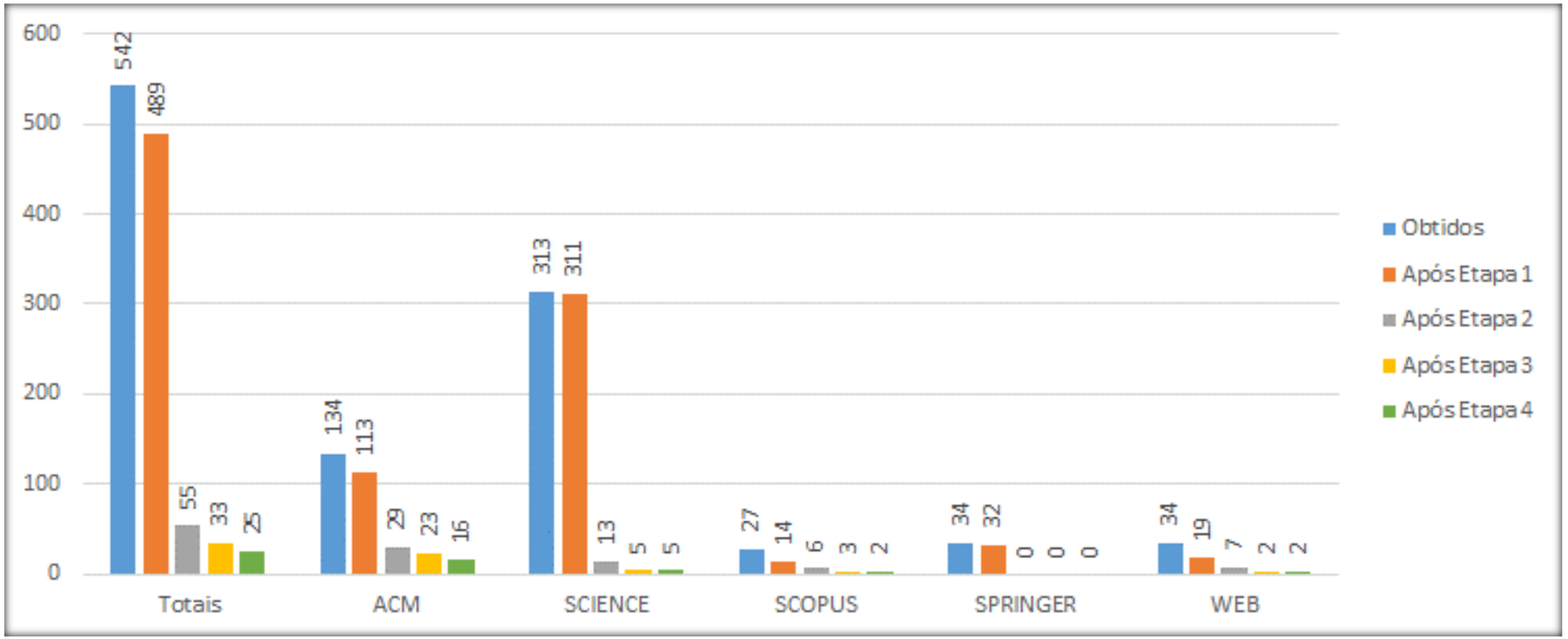
Etapa 2: análise dos títulos.

Etapa 3: análise dos resumos.

Etapa 4: aplicação dos demais critérios de inclusão e exclusão.

Inclusão	Exclusão
Artigos publicados em ciência da computação	Artigos duplicados
Artigos em inglês	Tutoriais, pôsteres, painéis, palestras, mesas redondas, oficinas, teses e dissertações
Estudos em formato eletrônico	Estudos que não podem ser localizados
Estudos publicados em conferências ou revistas	
Estudos relacionados ao tema da revisão	

Base de dados	Quantidade de estudos primários
ACM Digital Library	134
IEEE Xplore	0
Science Direct	313
Scopus	27
Springer	34
Web of Science	34
Total de estudos primários obtidos	**542**



Questões de pesquisa

QP1. Quais são as principais abordagens utilizadas na literatura para lidar com a interdependência entre as tarefas de alocação de registradores e escalonamento de instruções?

Abordagem cooperativa: as tarefas de alocação e escalonamento são realizadas separadamente, mas há uma certa troca de informação entre elas, no sentido de modo que uma pode levar em consideração as necessidades da outra.

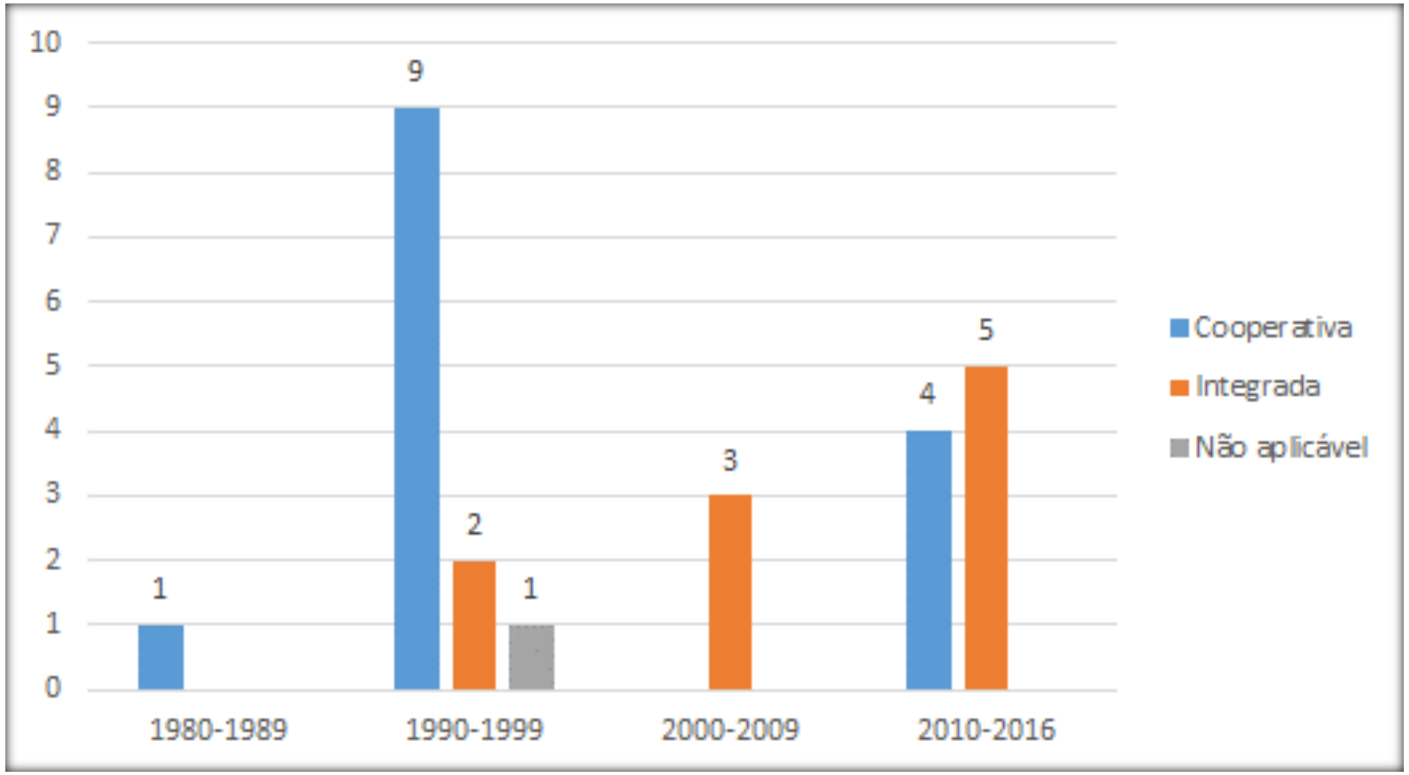
Abordagem integrada: as soluções para as duas tarefas são definidas simultaneamente.

Abordagem cooperativa: 56% dos estudos primários analisados.

Abordagem integrada: 40%.

Questões de pesquisa

QP1. Quais são as principais abordagens utilizadas na literatura para lidar com a interdependência entre as tarefas de alocação de registradores e escalonamento de instruções?



Questões de pesquisa

QP2. Abordagens que tratam as tarefas de alocação de registradores e escalonamento de instruções de forma conjunta melhoram a otimização do código?

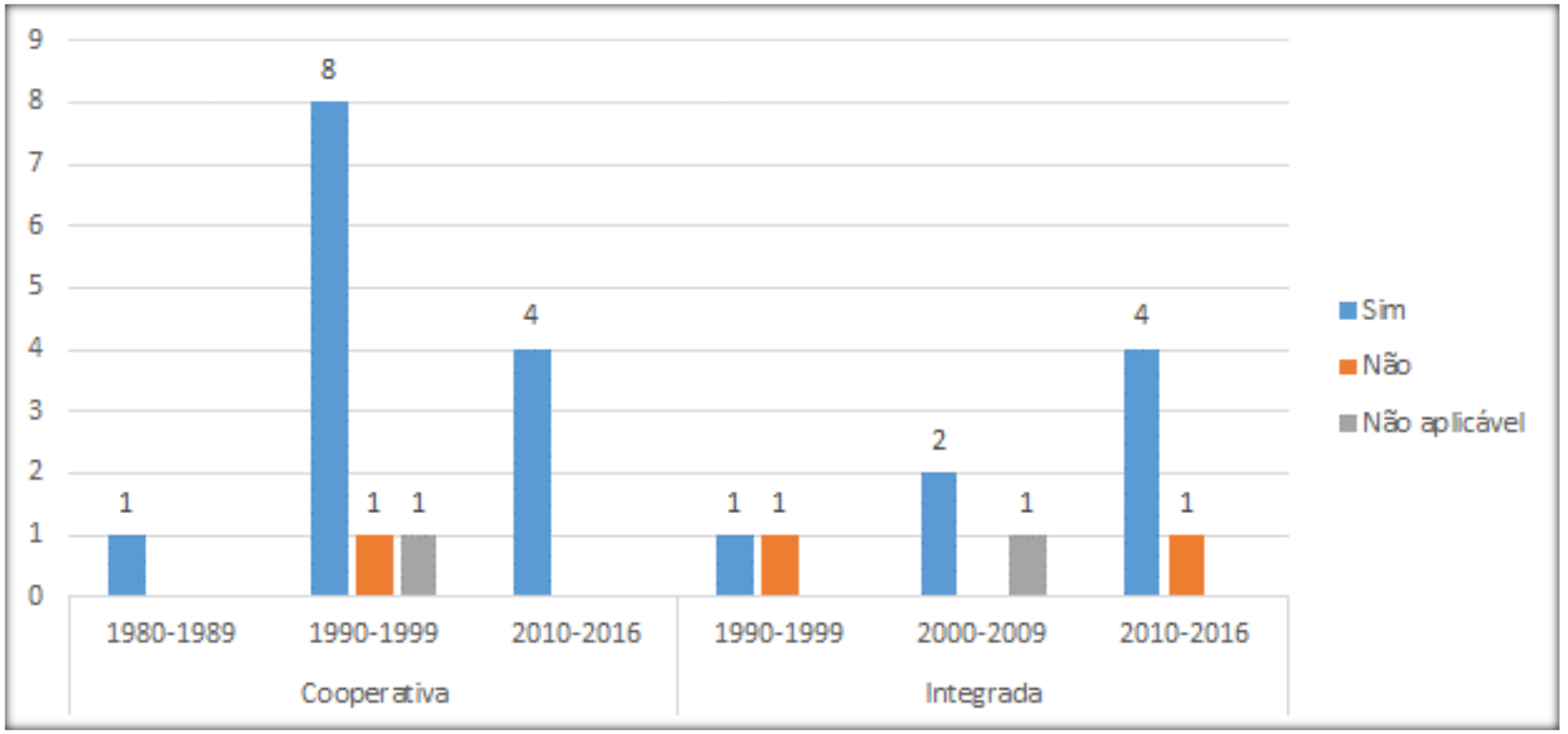
Sim.

Abordagens conjuntas melhoraram a otimização do código em 80% dos estudos primários avaliados.

Abordagem cooperativa: 93% dos estudos primários.
Abordagem integrada: 78%.

Questões de pesquisa

QP2. Abordagens que tratam as tarefas de alocação de registradores e escalonamento de instruções de forma conjunta melhoram a otimização do código?



Questões de pesquisa

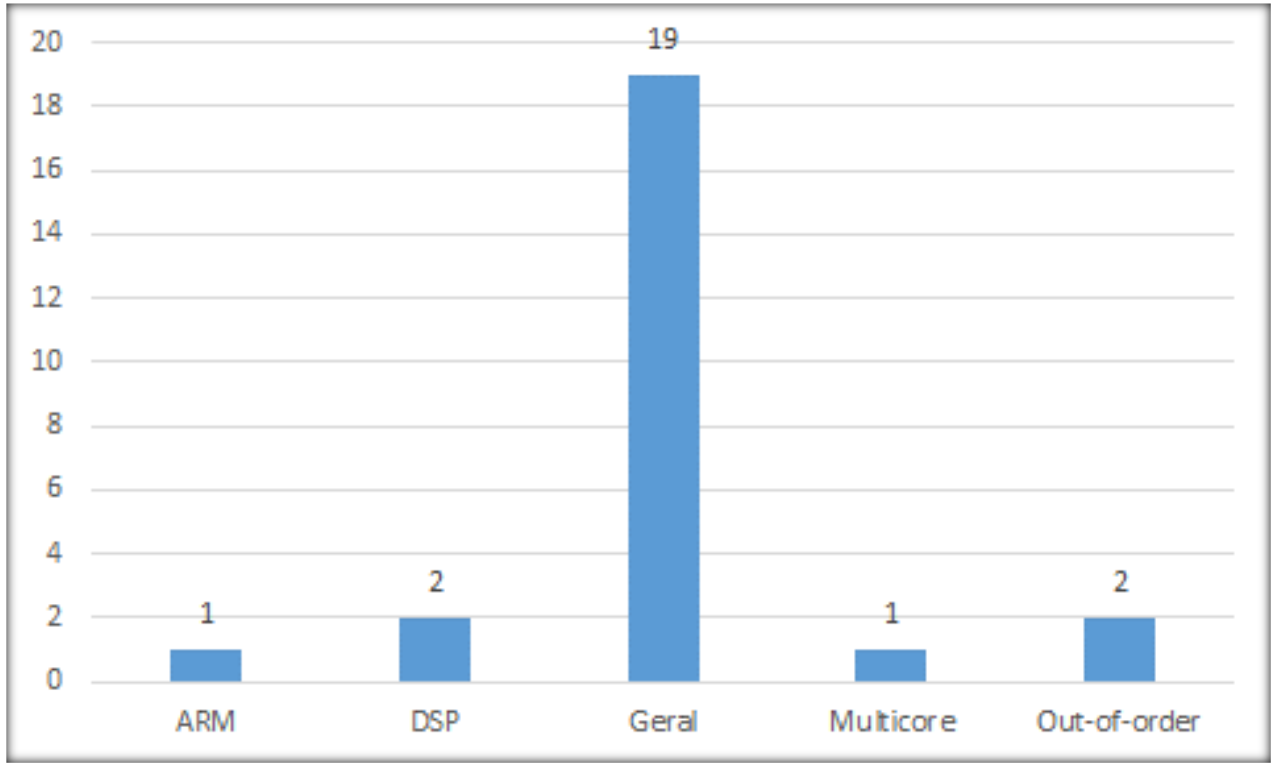
QP3. Para quais tipos de arquitetura as tarefas de alocação de registradores e escalonamento de instruções têm sido realizadas conjuntamente?

Arquiteturas do tipo RISC: 76% dos estudos primários analisados.

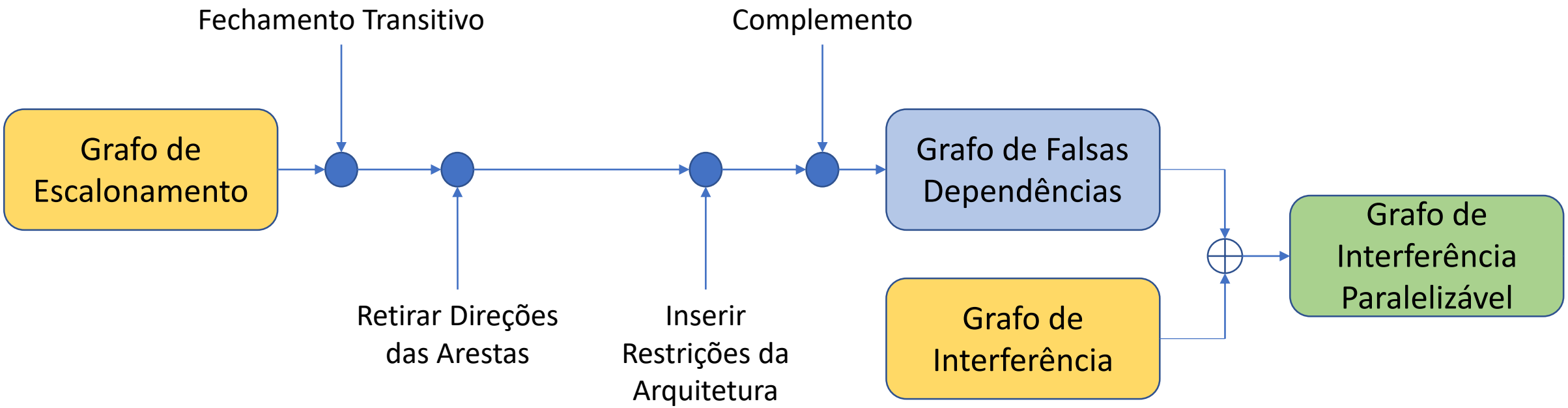
Outros tipos de arquiteturas: 24%.

Questões de pesquisa

QP3. Para quais tipos de arquitetura as tarefas de alocação de registradores e escalonamento de instruções têm sido realizadas conjuntamente?



Solução escolhida



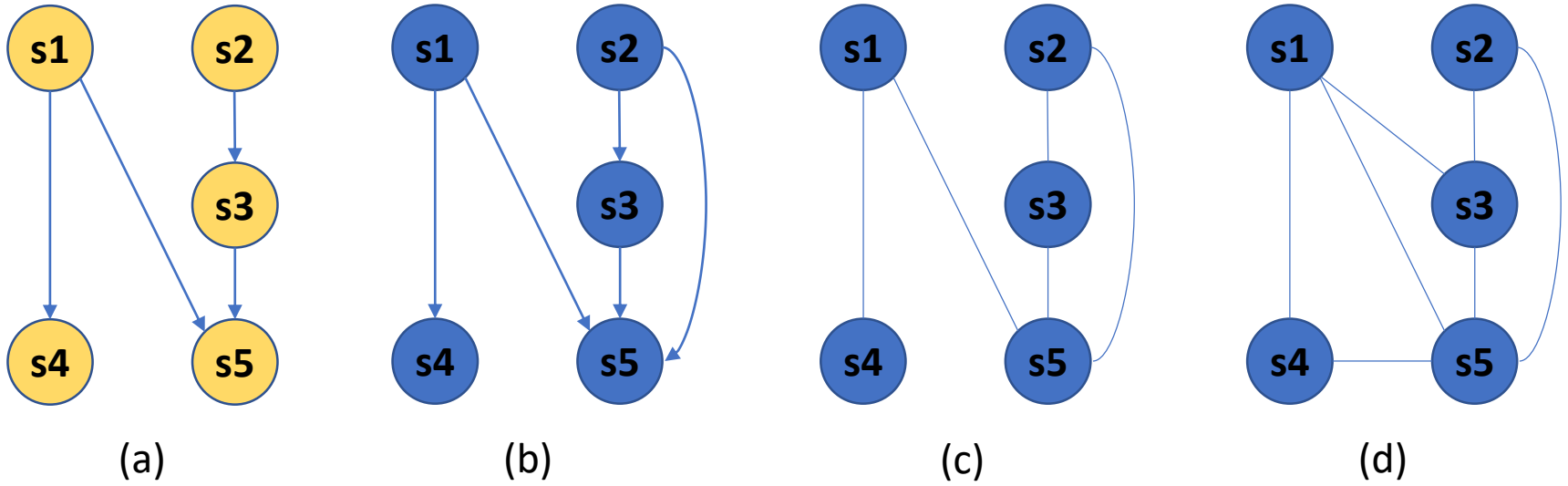
```
x := a[i]
y := z + z
z := x * 5 + z
```

Código fonte

```
s1 := load z
s2 := i
s3 := a[s2]
s4 := s1 + s1
s5 := s3 * 5 + s1
```

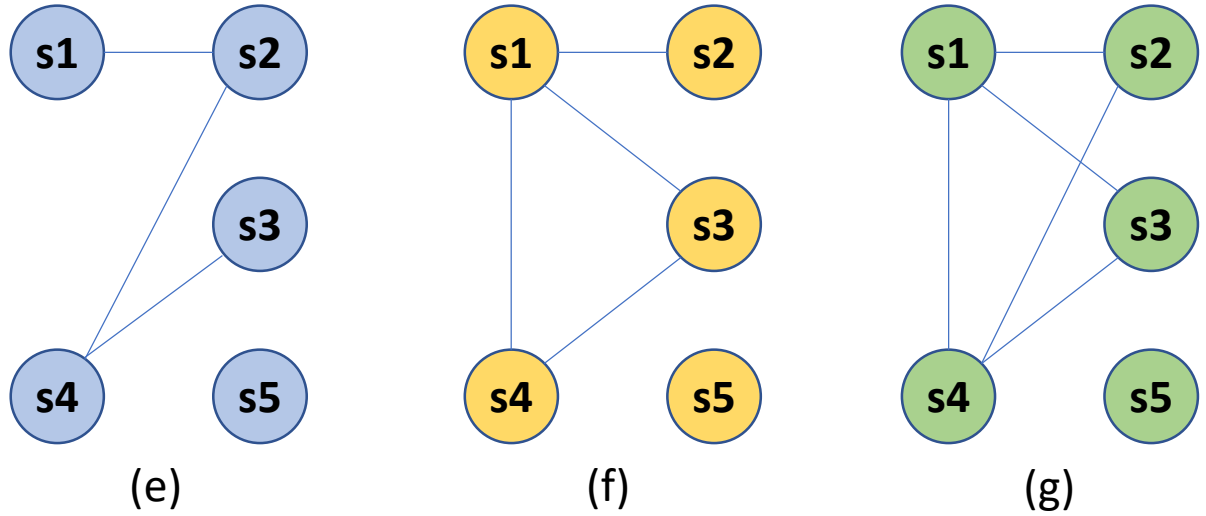
Código intermediário

Extraído de:
Shlomit Sarah Pinter
Register allocation with instruction scheduling: a new approach, 1996



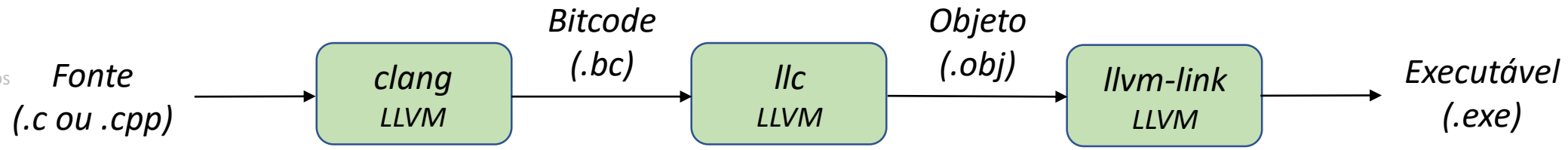

```
s1 := load z
s2 := i
s3 := a[s2]
s4 := s1 + s1
s5 := s3 * 5 + s1
```

Código intermediário

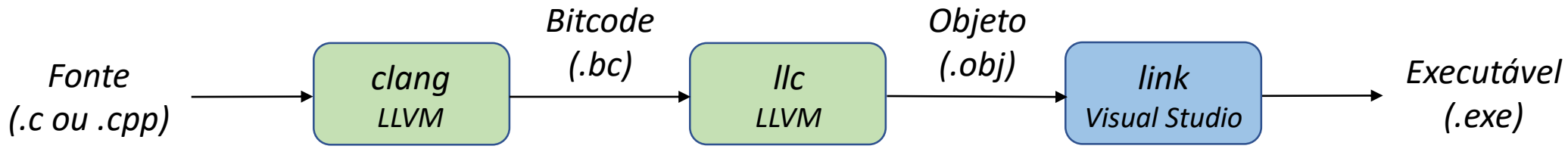


Implementação

Na plataforma Unix

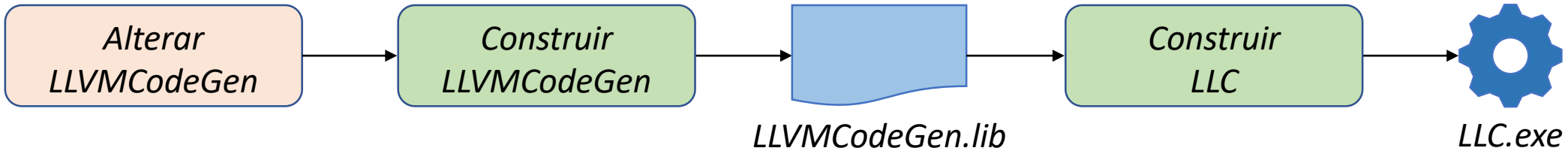


Na plataforma Windows

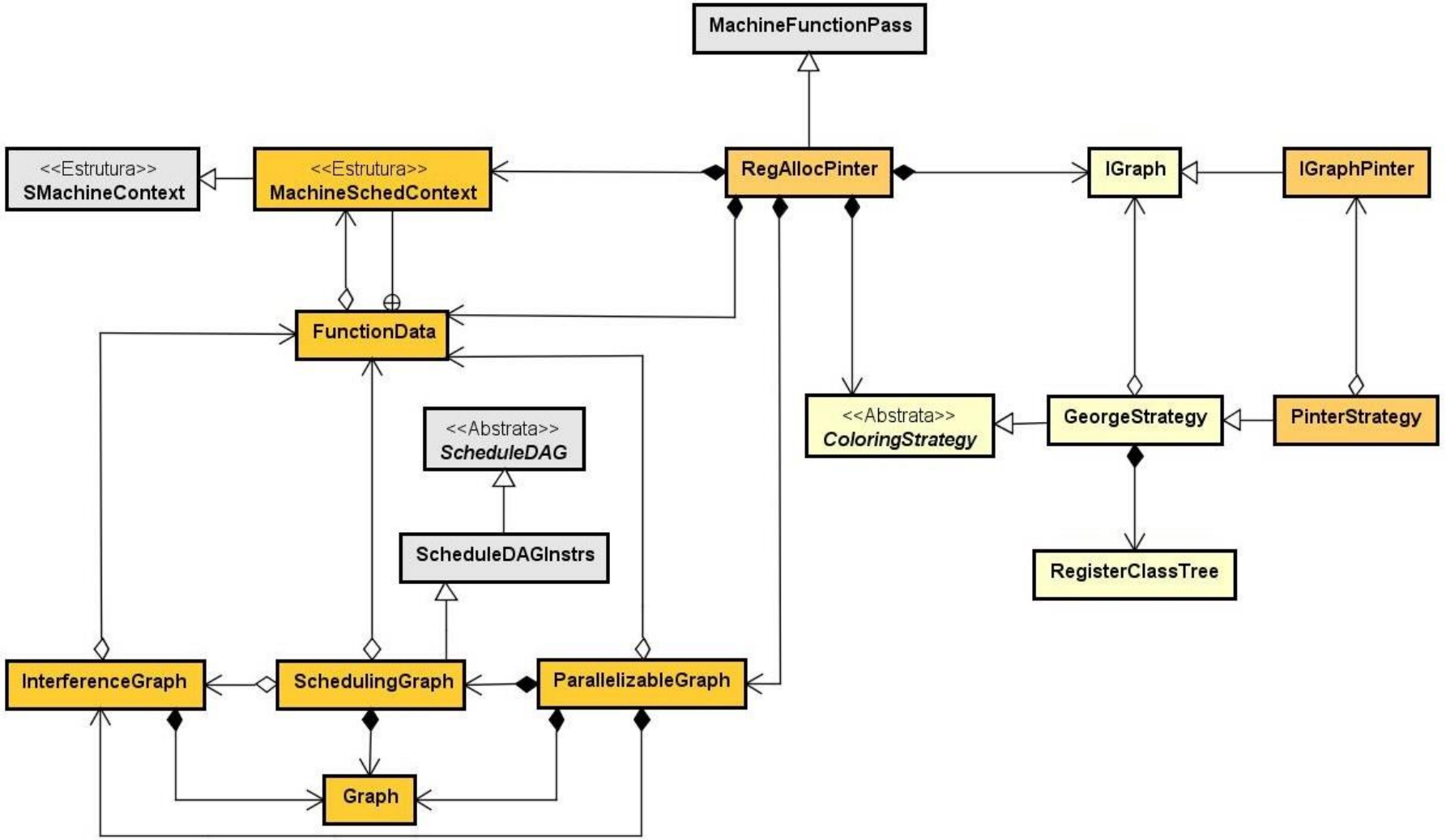


1. Introdução
2. Revisão Sistemática da Literatura
3. Solução escolhida
4. Implementação
   - Como compilar

   → Alterando o LLC
   - Código desenvolvido
   - Heurísticas H1 e H2
   - Heurística HB
   - Método ColorH2
5. Testes e análise dos resultados
6. Conclusão



llc.exe -regalloc=*basic*
llc.exe -regalloc=*greedy*
(...)
llc.exe -regalloc=*pinter*



1. Retirar do grafo os vértices que não podem ser derramados para a memória (custo de derramamento igual à constante *llvm::huge_valf*).

2. Retirar do grafo os vértices livres, aqueles que não se conectam a outros vértices.

3. (*Apenas para a heurística H1*) - Retirar do grafo o vértice de maior grau e menor número de registradores físicos associados.

3. (*Apenas para a heurística H2*) - Retirar do grafo o vértice com maior custo de derramamento e menor número de registradores físicos associados.

4. Sempre que um vértice é retirado do grafo, o grau dos demais vértices e o número de registradores físicos associados é atualizado. Os vértices retirados do grafo, cujo grau é menor do que o número de registradores físicos, são coloridos. Caso o vértice retirado do grafo não possa ser atribuído a nenhum registrador físico, ele é derramado para a memória.

1. Retira do grafo todos os vértices cujo grau é menor do que o número de registradores físicos disponíveis.

2. Restando vértices no grafo, o vértice com maior custo de derramamento é identificado e retirado do grafo.

3. Sempre que um vértice é retirado do grafo, o grau dos demais vértices e o número de registradores físicos associados é atualizado. Os vértices retirados do grafo, cujo grau é menor do que o número de registradores físicos, são coloridos. Caso o vértice retirado do grafo não possa ser atribuído a nenhum registrador físico, ele é derramado para a memória.

- Alocação de registradores utilizando a coloração do grafo de interferência (*tradicional*).

- Retirada dos vértices do grafo utilizando a heurística H2.

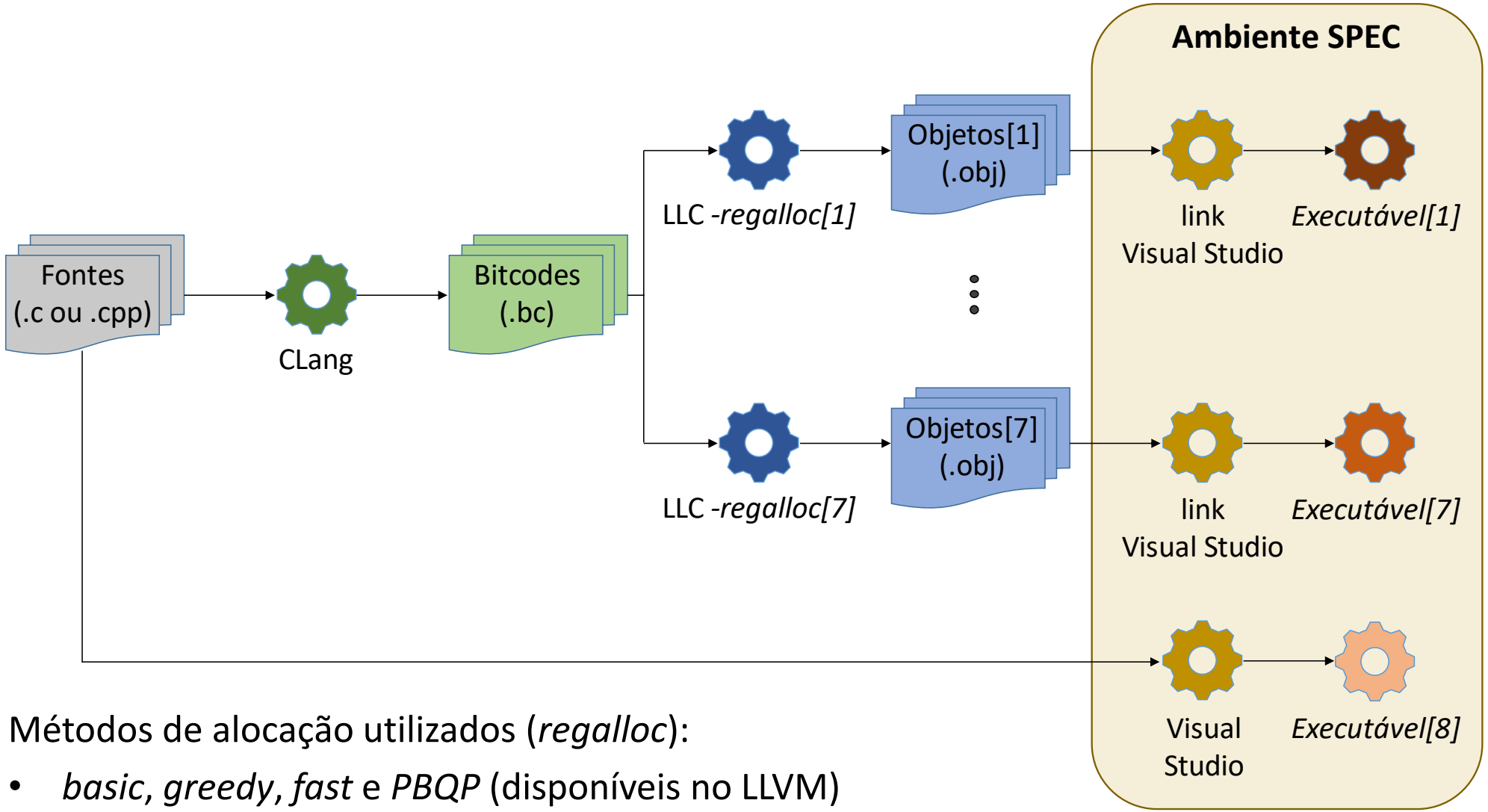
Testes e análise dos resultados

1. Testes realizados com os seguintes *benchmarks* do conjunto SPEC CPU2006:

Benchmark	Grupo	Linguagem
401.bzip2	CINT2006	C
429.mcf	CINT2006	C
433.milc	CFP2006	C
445.gobmk	CINT2006	C
458.sjeng	CINT2006	C
462.libquantum	CINT2006	C
464.h264ref	CINT2006	C
470.lbm	CFP2006	C
482.sphinx3	CFP2006	C
473.astar	CINT2006	C++

2. Execuções sob a plataforma Microsoft Windows.

3. Arquitetura X86.



Métodos de alocação utilizados (*regalloc*):

- *basic*, *greedy*, *fast* e *PBQP* (disponíveis no LLVM)
- *pinterHB*, *pinterH2* e *colorH2* (implementados)

T1: execução dos *benchmarks* compilados com os métodos existentes no LLVM, a fim de determinar qual deles possui melhor desempenho.

T2: execução dos *benchmarks* compilados com as heurísticas implementadas, determinando qual delas apresenta melhor desempenho.

T3: comparação dos resultados encontrados em T1 e T2.

T4: execução dos *benchmarks* compilados com o Microsoft Visual Studio e comparação com os resultados dos testes T1 e T2.

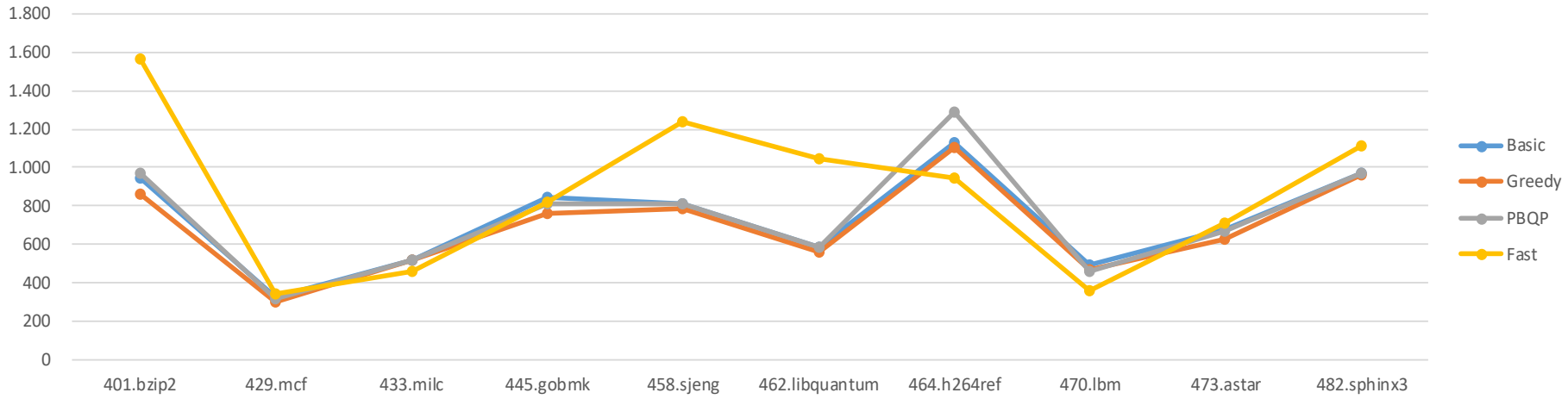
T5: comparações relacionadas ao derramamento de variáveis para a memória.

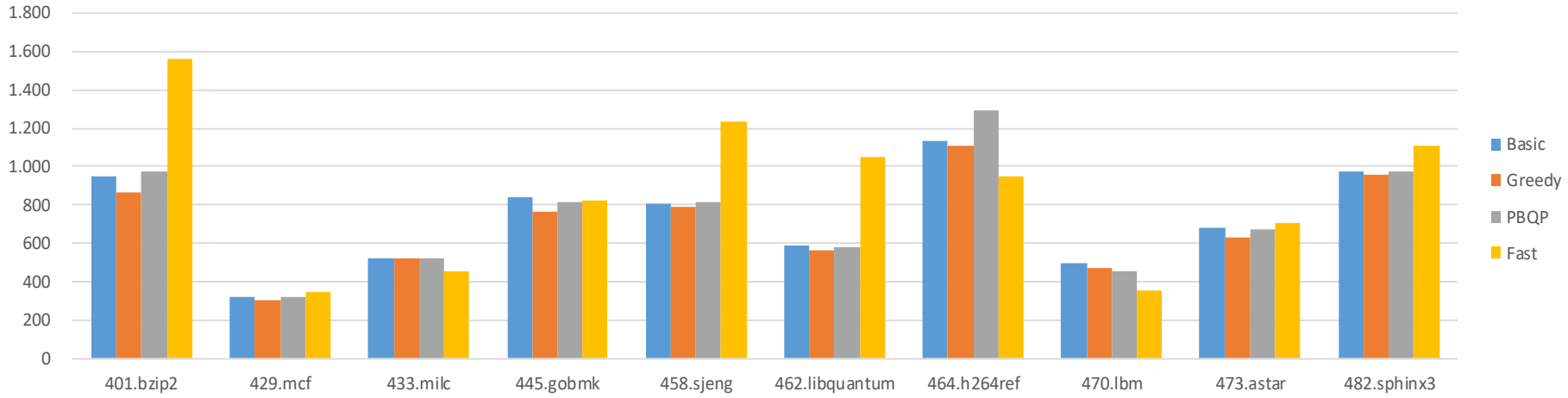
T6: comparações relacionadas aos esforços de compilação.

T7: avaliação do momento da execução da tarefa de escalonamento.

1. Introdução
2. Revisão Sistemática da Literatura
3. Solução escolhida
4. Implementação
5. Testes e análise dos resultados
   - Características
   - Compilação
   - *Testes definidos*
   - → Resultado do Teste T1
   - Resultado do Teste T2
   - Resultado do Teste T3
   - Resultado do Teste T4
   - Resultado do Teste T5
   - Resultado do Teste T6
   - Resultado do Teste T7
   - Análise
6. Conclusão

Benchmarks compilados com os métodos do LLVM



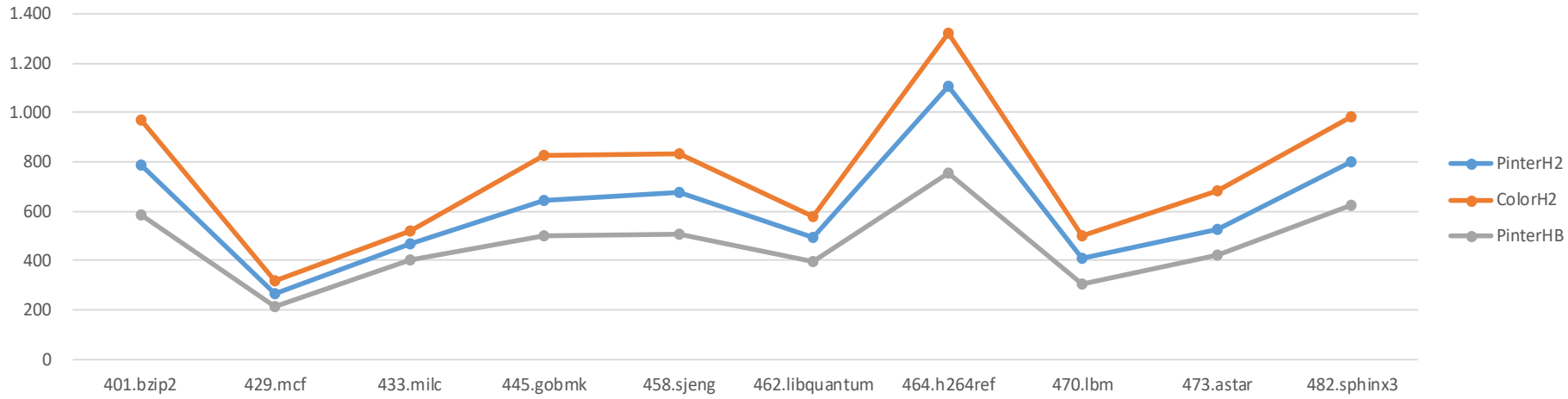


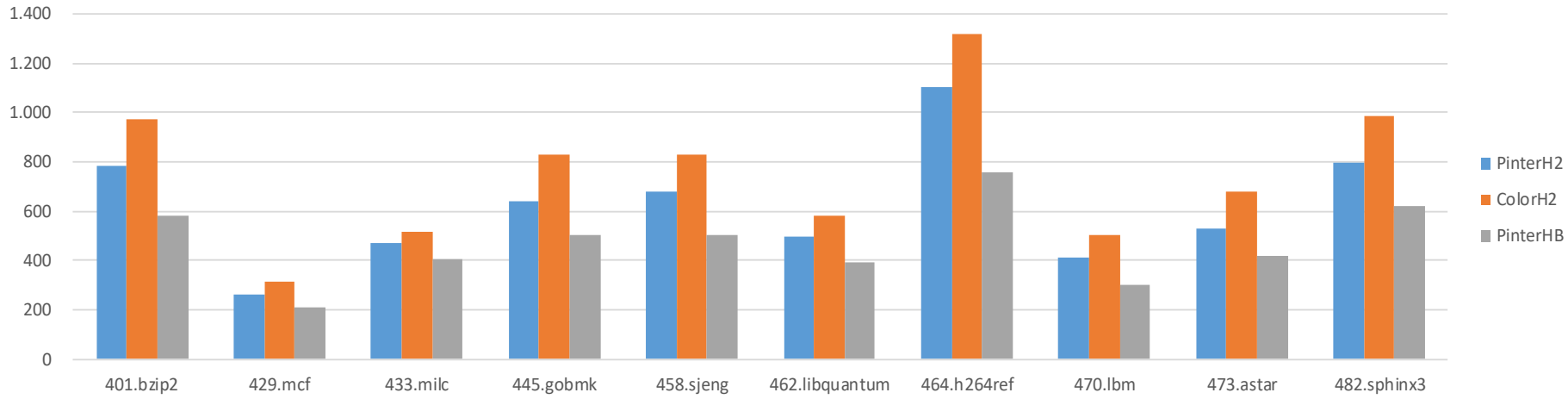
1. Introdução
2. Revisão Sistemática da Literatura
3. Solução escolhida
4. Implementação
5. Testes e análise dos resultados
   - Características
   - Compilação
   - *Testes definidos*
   - Resultado do Teste T1

   → Resultado do Teste T2

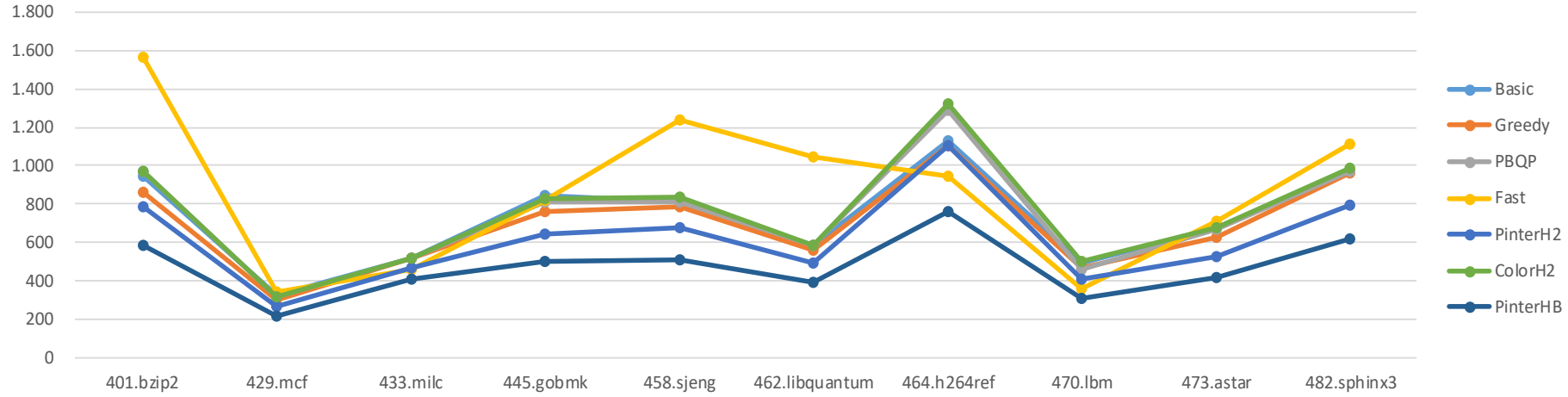
   - Resultado do Teste T3
   - Resultado do Teste T4
   - Resultado do Teste T5
   - Resultado do Teste T6
   - Resultado do Teste T7
   - Análise
6. Conclusão

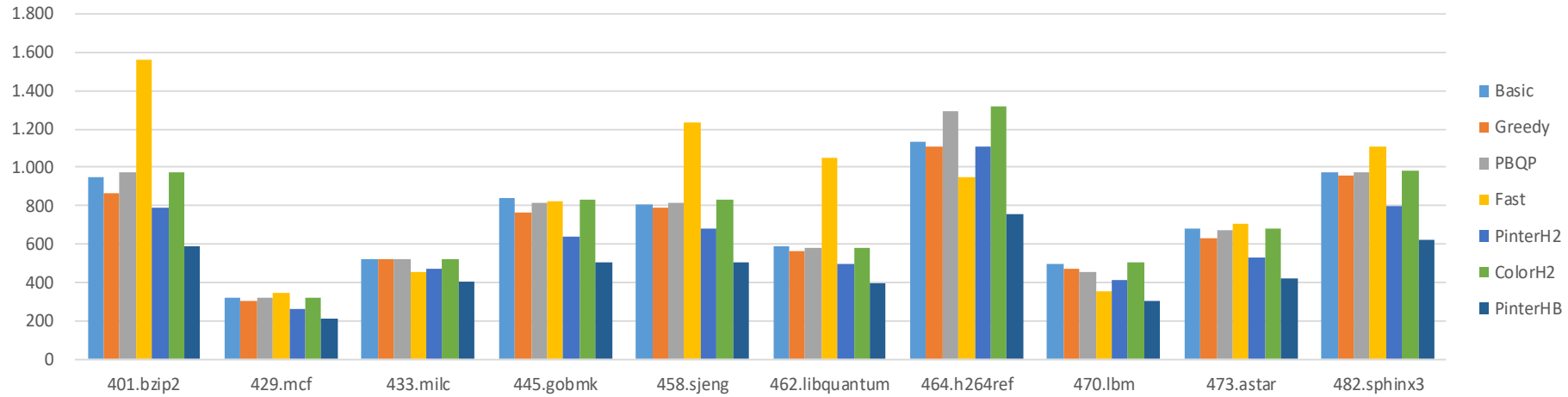
Benchmarks compilados com as heurísticas implementadas





Comparação entre o LLVM e as heurísticas implementadas



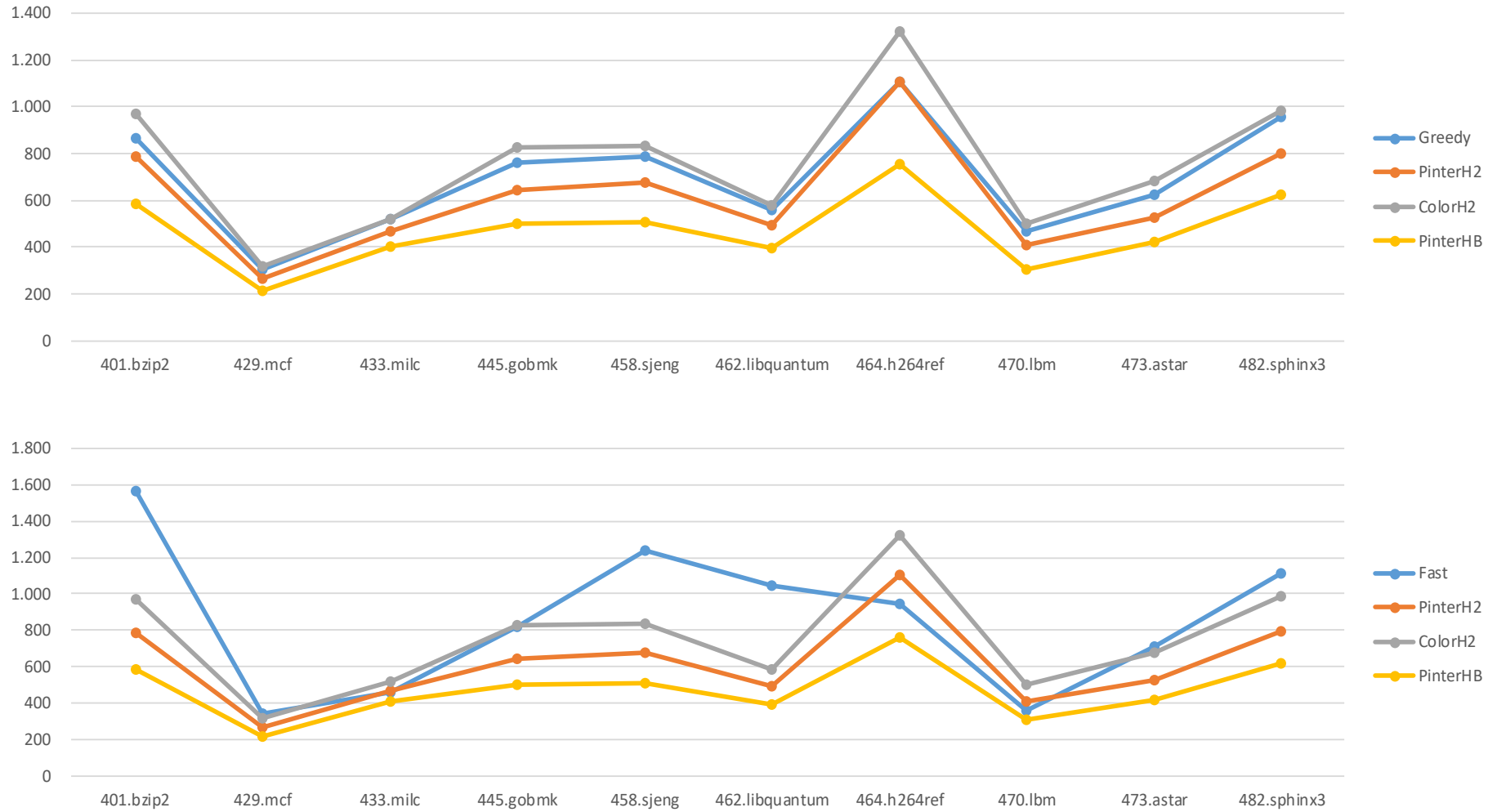


1. Introdução
2. Revisão Sistemática da Literatura
3. Solução escolhida
4. Implementação
5. Testes e análise dos resultados
   - Características
   - Compilação
   - *Testes definidos*
   - Resultado do Teste T1
   - Resultado do Teste T2

   → Resultado do Teste T3

   - Resultado do Teste T4
   - Resultado do Teste T5
   - Resultado do Teste T6
   - Resultado do Teste T7
   - Análise
6. Conclusão

Comparação entre o LLVM e as heurísticas implementadas



Resultados das comparação entre o LLVM e as heurísticas implementadas

	Ganhos (%)			
Método	**Mínimo**	**Máximo**	**Média**	**Desvio Padrão**
PinterHB	22,05	35,82	31,86	4,06
PinterH2	0,06	16,88	11,74	4,86
ColorH2	-19,28	-0,21	-7,30	5,49

Tabela 7.3. Ganhos da abordagem de Pinter sobre o método *Greedy* do LLVM.

	Ganhos (%)			
Método	**Mínimo**	**Máximo**	**Média**	**Desvio Padrão**
PinterHB	11,69	62,62	39,22	18,99
PinterH2	-16,74	52,78	21,15	25,41
ColorH2	-41,40	44,70	4,34	29,74

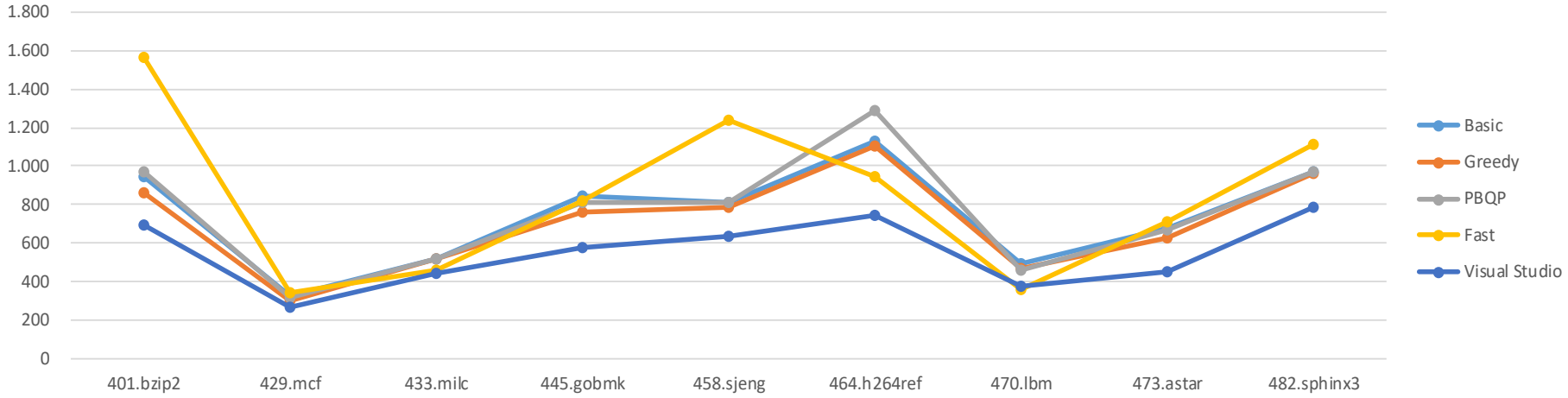
Tabela 7.4. Ganhos da abordagem de Pinter sobre o método *Fast* do LLVM.

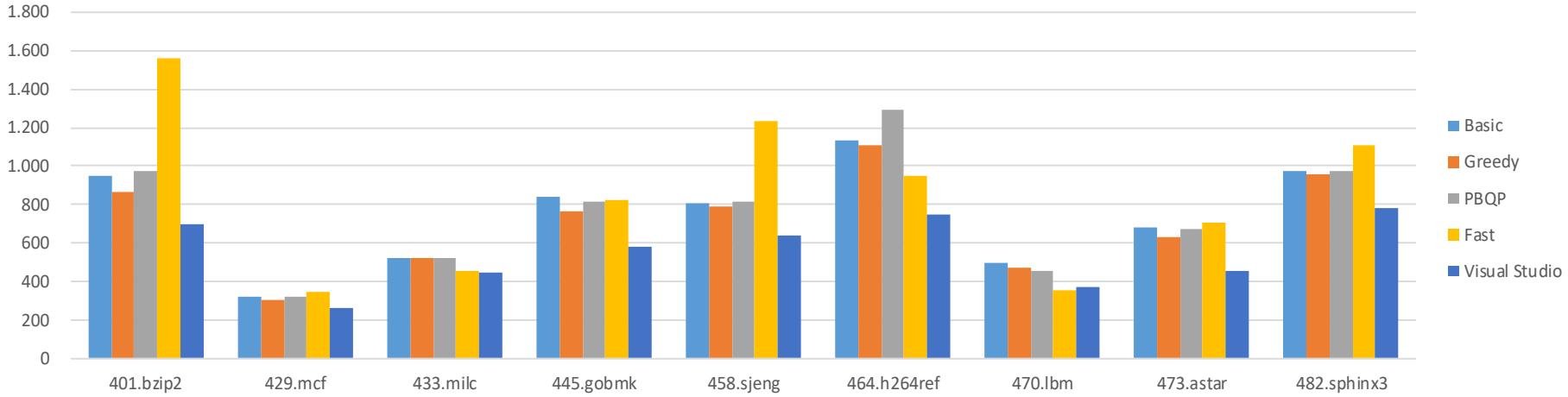
1. Introdução
2. Revisão Sistemática da Literatura
3. Solução escolhida
4. Implementação
5. Testes e análise dos resultados
   - Características
   - Compilação
   - *Testes definidos*
   - Resultado do Teste T1
   - Resultado do Teste T2
   - Resultado do Teste T3

   → Resultado do Teste T4

   - Resultado do Teste T5
   - Resultado do Teste T6
   - Resultado do Teste T7
   - Análise
6. Conclusão

Comparação entre o LLVM e o Visual Studio



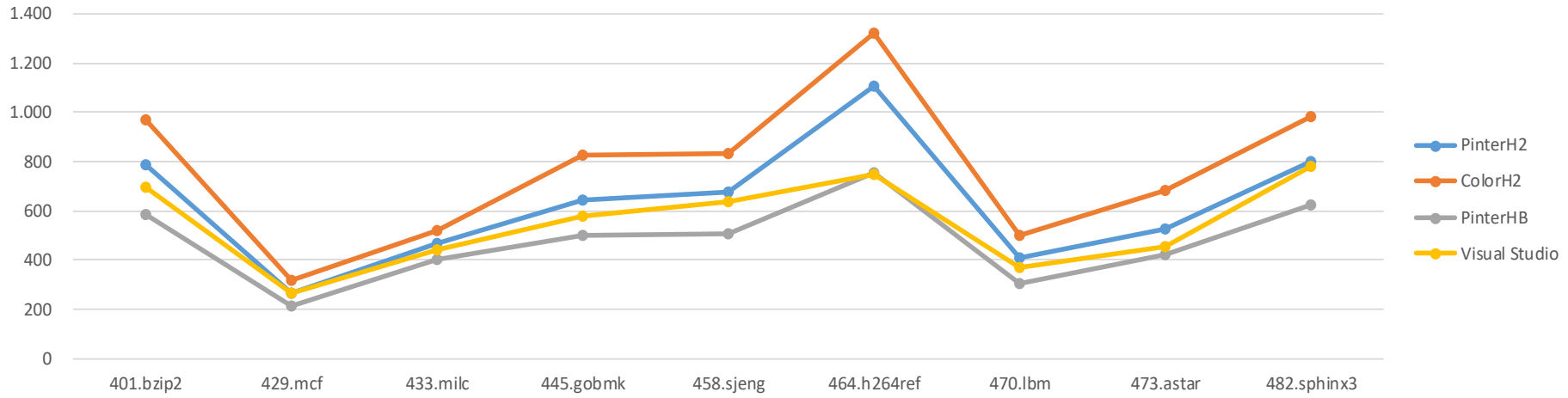


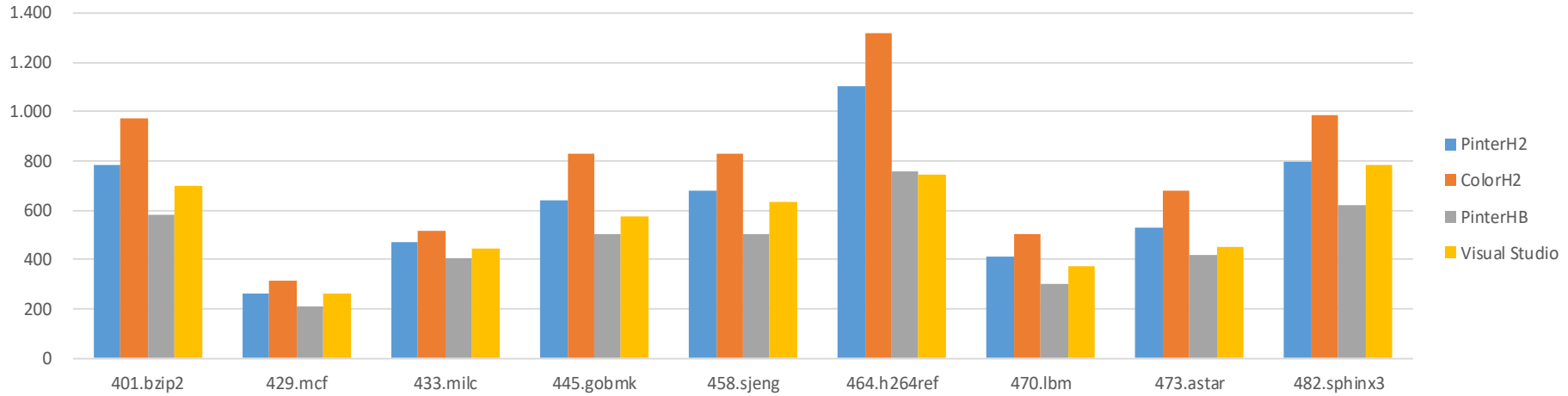
1. Introdução
2. Revisão Sistemática da Literatura
3. Solução escolhida
4. Implementação
5. Testes e análise dos resultados
   - Características
   - Compilação
   - *Testes definidos*
   - Resultado do Teste T1
   - Resultado do Teste T2
   - Resultado do Teste T3

   → Resultado do Teste T4

   - Resultado do Teste T5
   - Resultado do Teste T6
   - Resultado do Teste T7
   - Análise
6. Conclusão

Comparação entre as heurísticas implementadas e o Visual Studio





Resultados das comparações com o Visual Studio

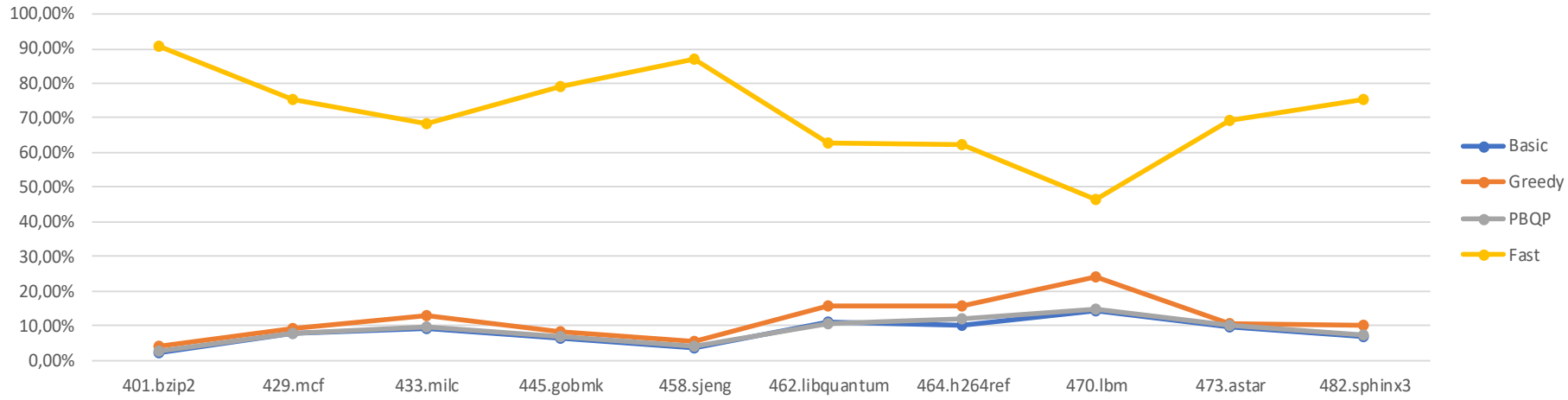
	Ganhos (%)			
Método	**Mínimo**	**Máximo**	**Média**	**Desvio Padrão**
PinterHB	-1,56	20,65	13,54	7,53
PinterH2	-48,36	-0,57	-12,75	14,32
ColorH2	-77,07	-17,21	-37,80	18,21

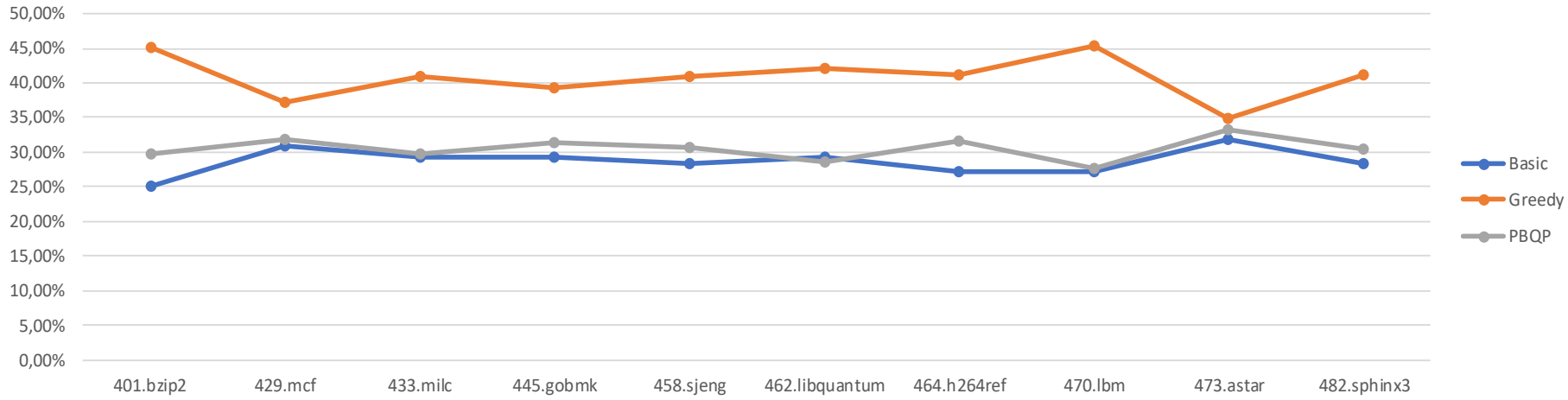
Tabela 7.6. Ganhos da abordagem de Pinter sobre o Visual Studio.

	Ganhos (%)			
Método	**Mínimo**	**Máximo**	**Média**	**Desvio Padrão**
Greedy	13,29	32,64	21,12	6,20
Fast	-4,53	55,41	27,00	19,29

Tabela 7.7. Ganhos do Visual Studio sobre os métodos do LLVM.

Derramamentos para a memória (LLVM)



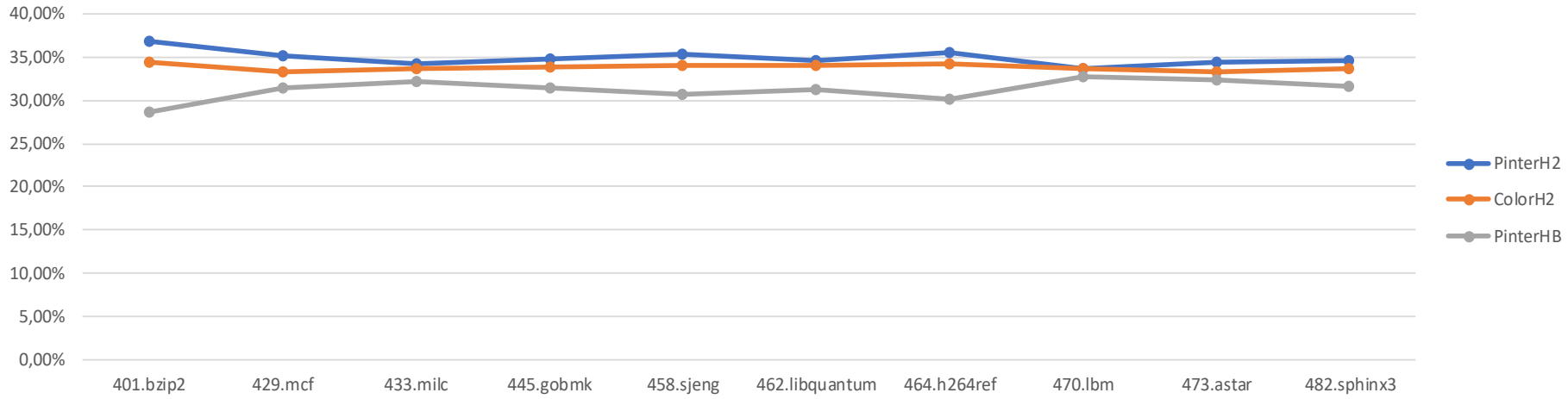


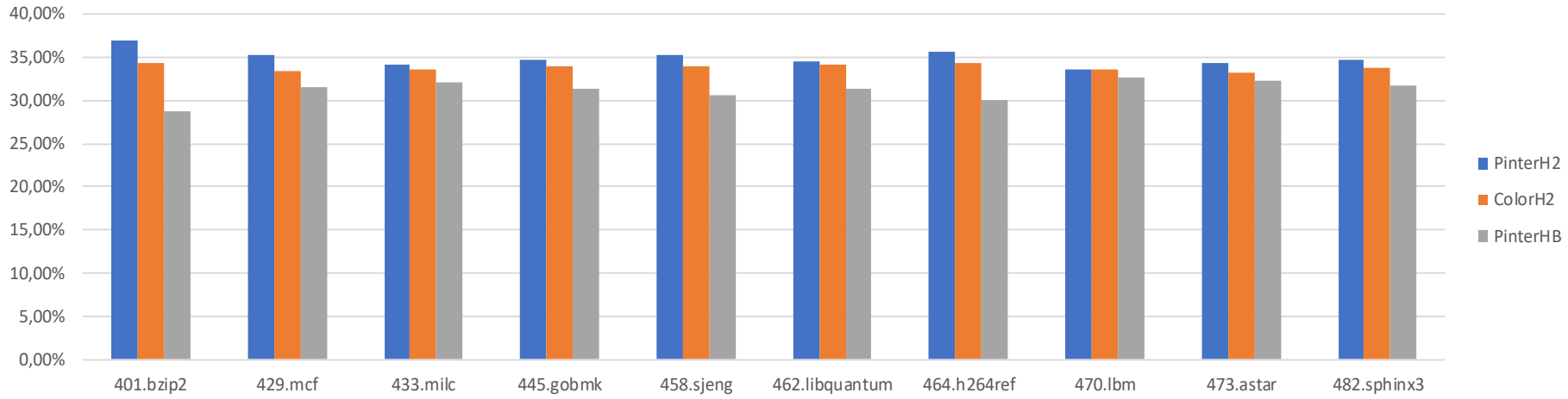
1. Introdução
2. Revisão Sistemática da Literatura
3. Solução escolhida
4. Implementação
5. Testes e análise dos resultados
   - Características
   - Compilação
   - *Testes definidos*
   - Resultado do Teste T1
   - Resultado do Teste T2
   - Resultado do Teste T3
   - Resultado do Teste T4

   → Resultado do Teste T5

   - Resultado do Teste T6
   - Resultado do Teste T7
   - Análise
6. Conclusão

Derramamentos para a memória (heurísticas implementadas)



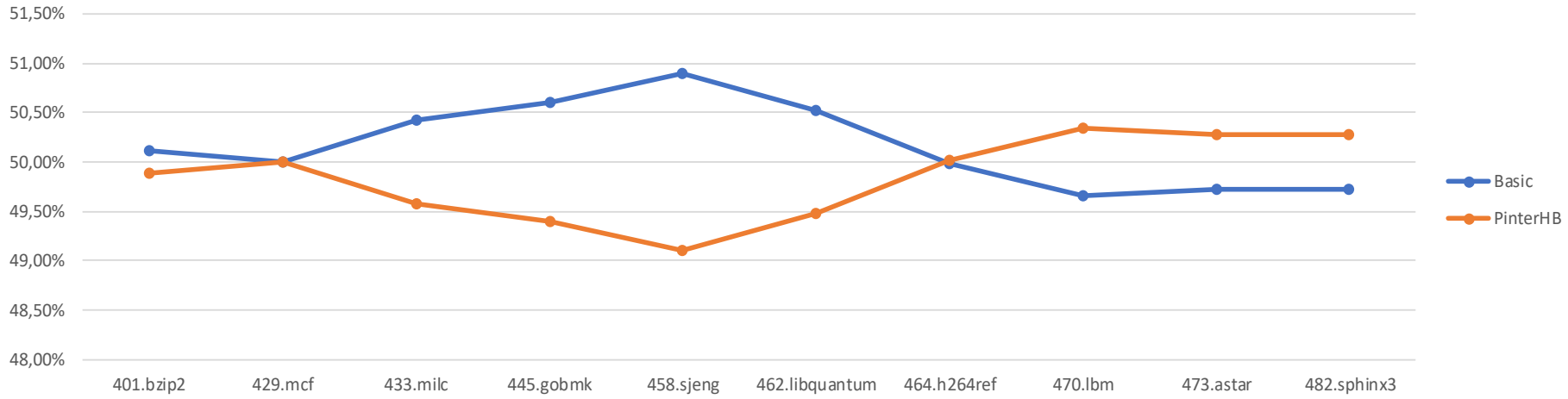


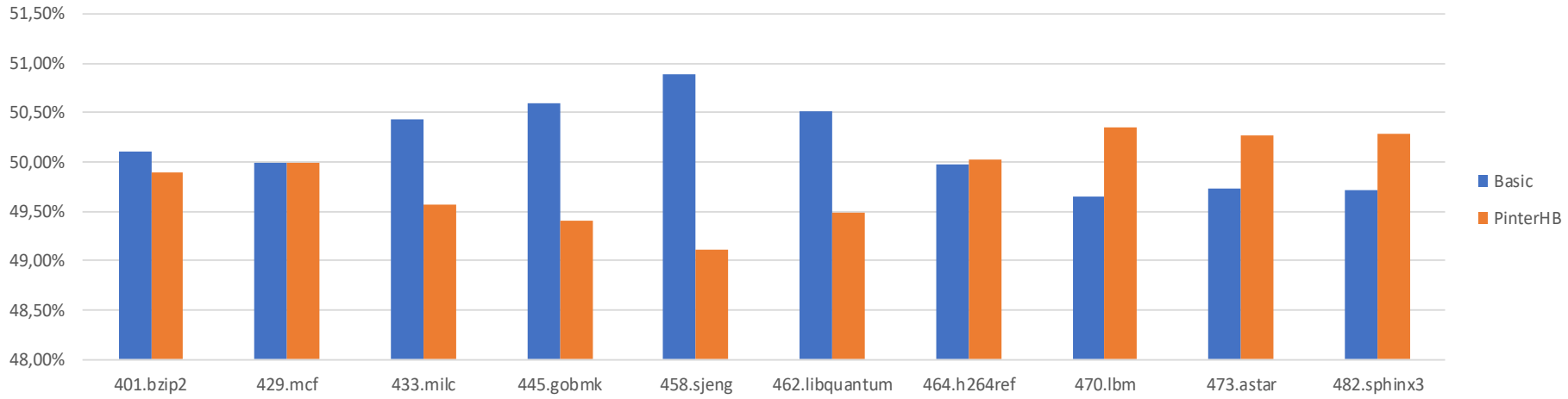
1. Introdução
2. Revisão Sistemática da Literatura
3. Solução escolhida
4. Implementação
5. Testes e análise dos resultados
   - Características
   - Compilação
   - *Testes definidos*
   - Resultado do Teste T1
   - Resultado do Teste T2
   - Resultado do Teste T3
   - Resultado do Teste T4

   → Resultado do Teste T5

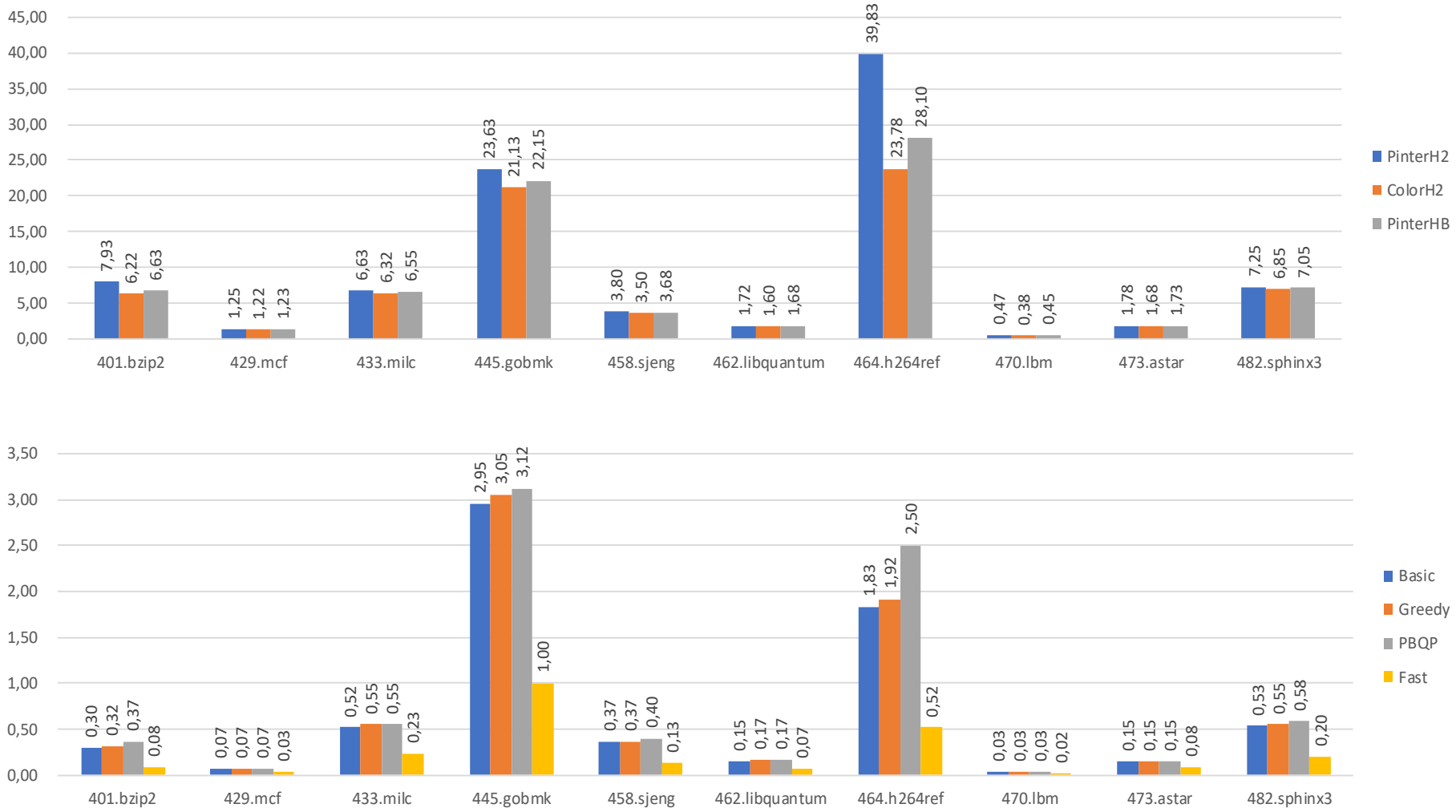
   - Resultado do Teste T6
   - Resultado do Teste T7
   - Análise
6. Conclusão

Derramamentos para a memória (comparação entre LLVM e heurísticas)





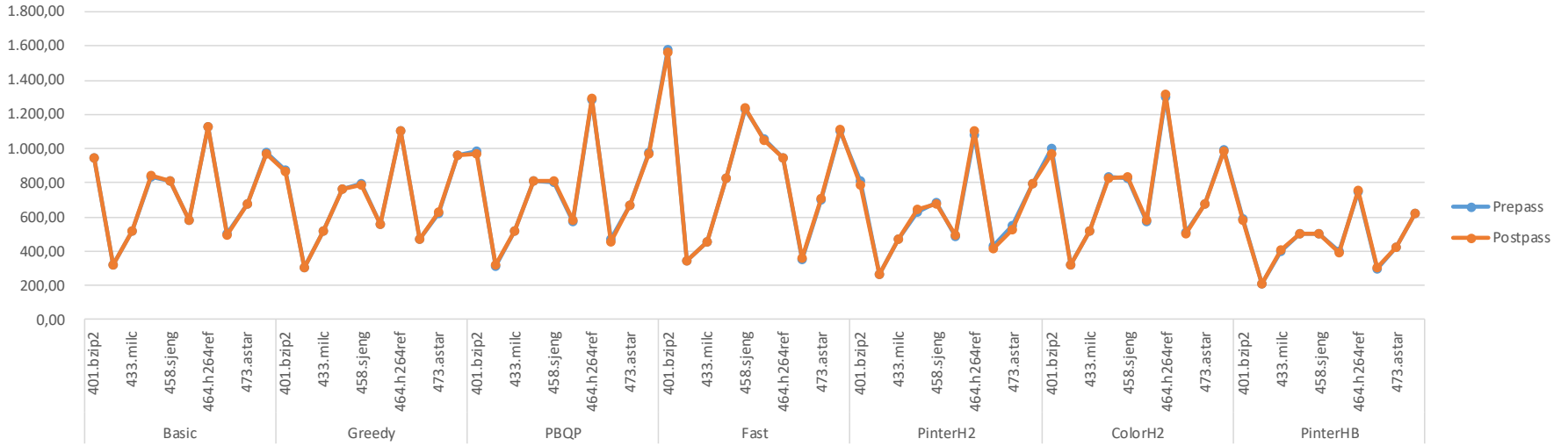
Tempos de compilação

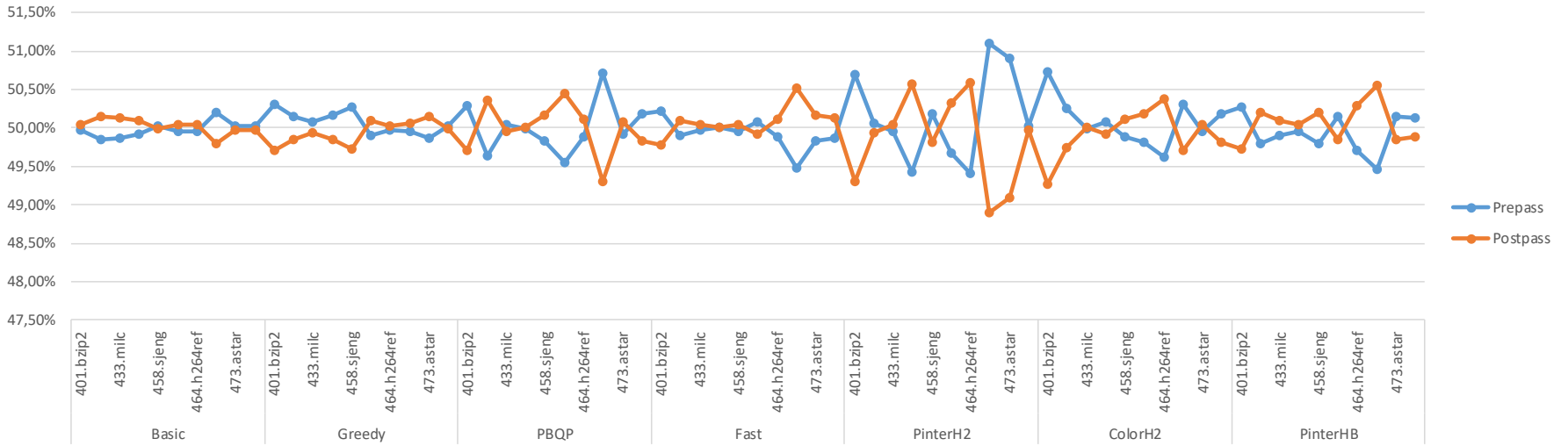


1. Introdução
2. Revisão Sistemática da Literatura
3. Solução escolhida
4. Implementação
5. Testes e análise dos resultados
   - Características
   - Compilação
   - *Testes definidos*
   - Resultado do Teste T1
   - Resultado do Teste T2
   - Resultado do Teste T3
   - Resultado do Teste T4
   - Resultado do Teste T5
   - Resultado do Teste T6

   → Resultado do Teste T7
   - Análise
6. Conclusão

Prepass X *Postpass*





1. A implementação da abordagem de Pinter utilizando a heurística HB para a coloração do grafo de interferência paralelizável foi capaz de gerar códigos eficientes.

2. Em função da arquitetura do microprocessador utilizado nos testes, a rigor, os resultados obtidos não podem ser relacionados unicamente ao tratamento dado ao problema da interdependência.

   Como demonstraram os resultados do Teste T7, as diferentes sequências de instruções definidas pelo compilador a partir dos escalonamentos *prepass* e *postpass* não produziram diferenças significativas sobre o desempenho dos programas testados, uma vez que o escalonamento final de cada um desses programas foi determinado dinamicamente pelo microprocessador, por meio da iniciação fora de ordem das instruções.

   Tendo em vista que não é possível conhecer o escalonamento final definido, nem mensurar o trabalho necessário para a sua produção, não há como afirmar de forma absolutamente precisa, com os testes realizados, que a implementação feita é eficiente para tratar o problema da interdependência entre as tarefas.

   Testes adicionais, realizados com arquiteturas que respeitam a ordem de instruções definida pelo compilador, tais como as arquiteturas da maioria dos microprocessadores ARM, seria possível comprovar categoricamente a eficiência da implementação desenvolvida como solução para o problema.

3. Os testes realizados fornecem indícios de que a implementação feita está relacionada a uma boa e adequada solução para o problema:

i. Um *primeiro* indício está associado aos resultados de desempenho.

Código ruim implica muitos derramamentos de valores para a memória, ou muitas falsas dependências, ou ambos.

Ainda que o escalonamento dinâmico fosse rápido o bastante para compensar a perda de desempenho associada aos derramamentos para a memória e bom o bastante para eliminar as falsas dependências existentes no código, seria de se esperar que o tratamento inadequado do problema gerasse resultados ruins, ou ao menos não tão bons quanto os que foram observados.

Seria de se esperar que o tamanho total do escalonamento aumentasse em algumas situações, impactando o desempenho dos programas relacionados.

Mas, dentre os 70 códigos produzidos com o escalonamento *postpass*, todos aqueles que apresentaram melhor desempenho foram compilados com o método PinterHB.

ii. Um *segundo* indício está relacionado aos derramamentos de valores para a memória.

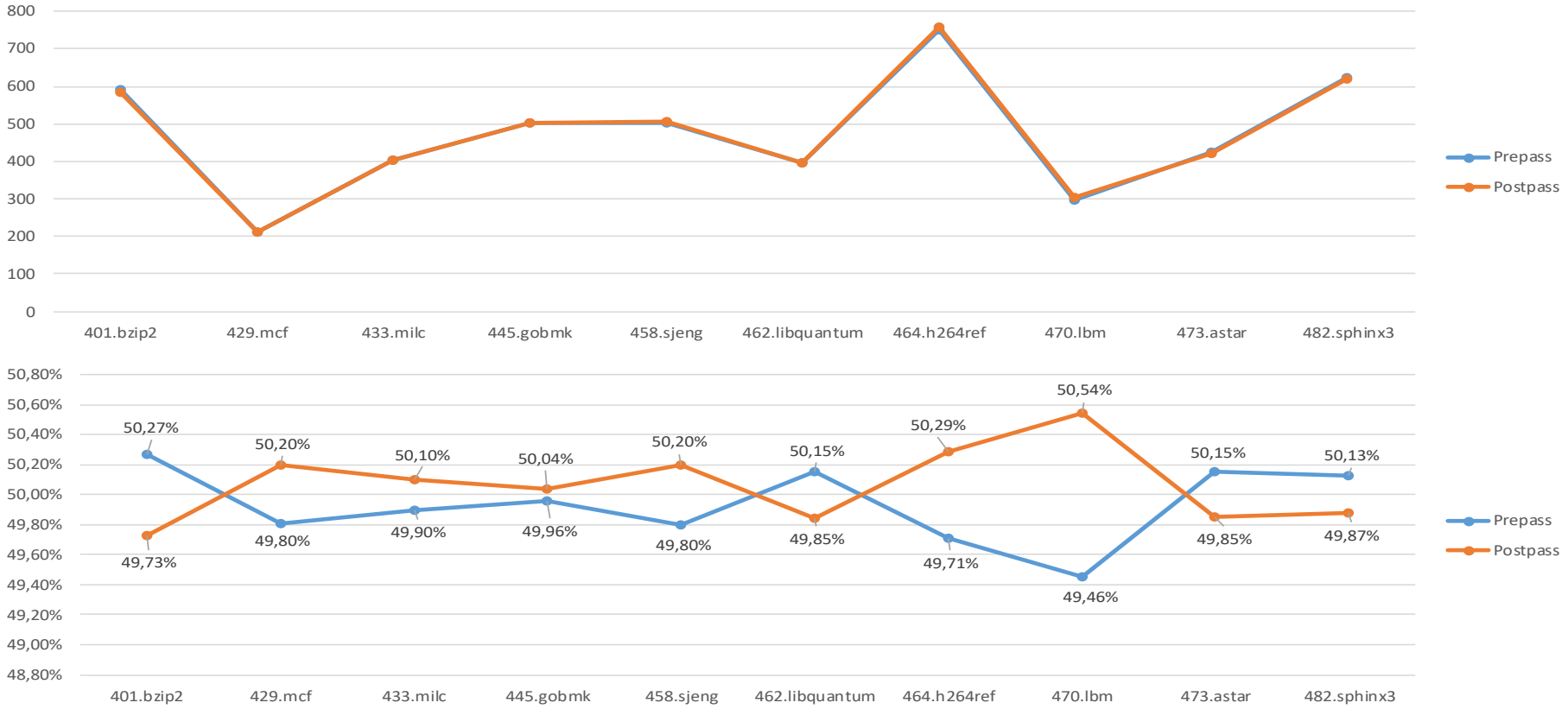
Como mostra o Teste T5, os bons resultados de desempenho fornecidos pelo método PinterHB não podem ser atribuídos apenas ao reduzido número de derramamentos produzidos.

iii. Um *terceiro e último* indício diz respeito ao valor médio relativo e à variância dos desempenhos dos códigos compilados com os escalonamentos *prepass* e *postpass* e com o método PinterHB.

Assumindo como verdade que o escalonamento dinâmico produz um código tão eficiente quanto possível, suposição absolutamente viável, pois trata-se de escalonamento feito por hardware, responsável em grande parte pelo maior desempenho das arquiteturas superescalares, e considerando os dois gráficos que se seguem, é possível constatar que o escalonamento dinâmico não alterou de modo significativo o código produzido com os escalonamentos *prepass* e *postpass*.

Isso indica que o escalonamento produzido pelo compilador foi de boa qualidade em ambos os casos. Se não fosse assim, no gráfico citado, deveríamos observar um desvio bem maior entre qualquer uma das duas curvas e a reta que define o ponto médio entre elas, isto é, a reta que corta o eixo das ordenadas no valor de 50%.

iii. Um *terceiro e último* indício diz respeito ao valor médio relativo e à variância dos desempenhos dos códigos compilados com o escalonamento postpass e com o método PinterHB.



4. A intenção dos indícios destacados não é apontar a implementação feita da abordagem de Pinter como solução evidente para o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções.

   A identificação desses indícios intenciona tão somente mostrar que a implementação feita pode ser tomada como uma abordagem capaz de tratar adequadamente esse problema, fornecendo uma solução de qualidade em termos do desempenho do código gerado.

5. Finalmente, é importante analisar a implementação feita a partir dos tempos de compilação observados (Teste T6).

   Tais tempos inviabilizam a utilização do método desenvolvido em ambientes de produção contínua de software, isto é, em situações de desenvolvimento nas quais as aplicações desenvolvidas são compiladas diversas vezes ao longo do desenvolvimento.

   Porém, em função dos resultados obtidos, o desenvolvimento realizado poderia ser utilizado como método de compilação final de quaisquer sistemas, principalmente, no caso de sistemas embarcados.

Conclusão

1. O problema da interdependência, apesar de antigo, tem sido objeto de estudos atuais.

2. O tratamento desse problema pode ter consequências muito positivas no processo de geração de códigos eficientes:

   - Os códigos compilados com o método implementado tiveram desempenho superior aos códigos gerados com o LLVM (média de 32%) e com o Visual Studio (média de 14%).

   - A abordagem implementada gerou códigos mais eficientes mesmo para arquiteturas CISC utilizando tecnologias superescalares.

   - Espera-se que a implementação feita também proporcione resultados positivos em arquiteturas RISC, ou em arquiteturas que não utilizem técnicas de iniciação fora de ordem de instruções.

1. Estudo em larga escala sobre o problema da interdependência entre as tarefas de alocação de registradores e escalonamento de instruções.

2. Artigo técnico Carvalho et al. [2017], intitulado The Register Allocation and Instruction Scheduling Challenge, apresentado no VIII CBSoft, XXI SBLP.

3. Exemplo real de implementação de um passo de alocação de registradores no LLVM, capaz de ser auxiliar e facilitar futuras implementações.

4. Implementação da abordagem de Pinter no LLVM.

5. Biblioteca em C++ para a manipulação de grafos simples, multigrafos e pseudografos, sejam eles dirigidos, ou não dirigidos.

Revisão Sistemática da Literatura

1. Alterar o protocolo de revisão, considerando novos termos de busca e novas bases de dados.

2. Definir novos critérios de análise dos estudos primários:
   - Quão ótimas são as soluções apresentadas?
   - Quais são as áreas e aplicações consideradas nos estudos?
   - Qual é a modelagem utilizada na solução?
   - Qual é o tempo de execução dos códigos desenvolvidos?
   - Qual é o número de softwares e trabalhos acadêmicos influenciados?

3. Comparar tecnicamente as diferentes abordagens selecionadas, para identificar as razões que fazem uma abordagem ser melhor do que outra.

4. Quantificar o impacto de cada estudo primário sobre os trabalhos subsequentes.

Implementação da abordagem de Pinter no LLVM

1. Realizar testes em arquiteturas RISC, ou em arquiteturas que não utilizem a iniciação fora de ordem de instruções.
   - Quaisquer modelos de quaisquer famílias de processadores ARM, tais como ARM7, ARM9, ARM11, Cortex-M, Cortex-R e Cortex-A, com exceção do modelo Cortex-A15 MPCore, podem ser utilizados.
2. Considerar análises de dominância e pós-dominância para refinar o processo de geração do grafo de interferência paralelizável.
3. Aprimorar o tratamento das irregularidades das arquiteturas.
4. Tornar as heurísticas desenvolvidas mais robustas, aprimorando o tratamento dado ao código *rematerializado*.
5. Implementar e testar novos métodos de escalonamento de instruções.

Questões

Muito obrigado!

João Francisco Neiva de Carvalho
joaofnc@dcc.ufmg.br