Técnicas de Otimização de Código Baseado em Máquinas de Estado Abstratas

Fabio Tirelo
Roberto da Silva Bigonha

LLP - DCC - UFMG

Máquinas de Estado Abstratas

- Máquinas de Estado Abstratas (ASM):
  - modelo formal criado por Y. Gurevich;
  - permitem a execução direta de especificações;
  - facilidade para escrever especificações em quaisquer níveis de abstração;
  - Já utilizadas na definição de arquiteturas, semântica de linguagens de programação, sistemas distribuídos, etc.

Eficiência de Programas ASM

- Limitações no modelo tornam os programas ineficientes se comparados a programas escritos em linguagem imperativa
- É necessário, portanto, o desenvolvimento de técnicas de otimização que ataquem diretamente os focos de ineficiência do modelo

Implementações Existentes

- Nenhuma das implementações existentes utilizam técnicas de manipulação de código específicas para ASM
- As únicas técnicas aplicadas referem-se à representação de funções e a eliminação de sub-expressões comuns.

Modelo ASM

- Um programa ASM consiste na definição de uma álgebra, ou estado inicial, e de uma regra de transição.
- A álgebra consiste na definição do conjunto de símbolos utilizado pelo programa - vocabulário.
- A regra de transição promove a mudança de estados (álgebras) por meio da modificação da interpretação dos símbolos do vocabulário.

Modelo ASM...

- Em ASM, sistemas são definidos operacionalmente por meio das mudanças de estado causadas por sua regra de transição
- No modelo, existe um estado atual *S* e um novo estado é criado executando-se a regra de transição em *S*

Modelo ASM...

- Uma regra de transição pode ser de três tipos:
  - *Atualização:* "$f(x) := y$". Tem o efeito de modificar a função f no ponto x para retornar y no próximo estado.
  - *Condicional*: "**if** g **then** R_1 **else** R_2 **end**", onde g é uma expressão booleana e R_1 e R_2 são regras de transição. Executa R_1 se g for verdadeira e R_2 caso contrário.
  - *Bloco:* "$R_1, R_2, ..., R_n$". Tem o efeito de executar as regras R_i em paralelo.

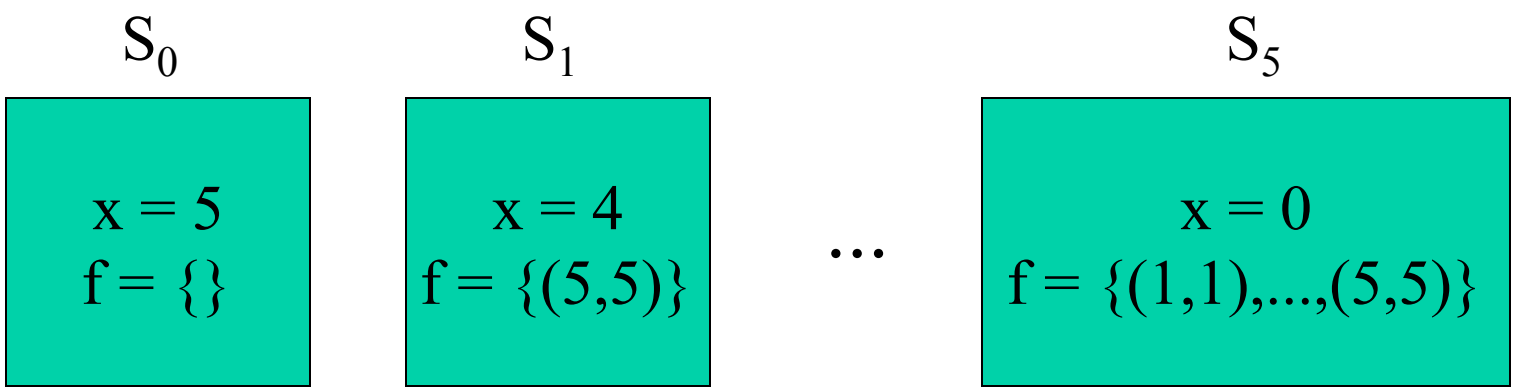
Modelo ASM...

- A execução de um programa ASM consiste na execução de sua regra de transição *repetidamente*, sendo que o efeito da execução de uma atualização em um passo só é percebido no passo seguinte.

Modelo ASM...

- Definição da função f = {(1,1), (2,2), ..., (k,k)}, onde k é o valor de x no estado inicial.

```
if x > 0 then
      x := x - 1,
      f(x) := x
end
```



Modelo ASM...

- Um endereço é um par (f, x), onde f é um nome de função n-ária pertencente ao vocabulário e x é uma tupla de n elementos pertencente ao domínio de f.
- Uma atualização é um par (l, y), onde l é um endereço e y é um elemento do contradomínio de f.

Modelo ASM...

- A regra de transição R, ao ser executada no estado S, gera um conjunto das atualizações feitas pela regra de transição, $Updates(R,S)$.
- A execução da regra de transição cria um novo estado S', no qual uma função f no ponto $(a_1, ..., a_n)$ é dado por:

$$S'(f(a_1,\ldots,a_n)) = \begin{cases} y & \text{se } ((f,(a_1,\ldots,a_n)),y) \in Updates(R,S) \\ S(f(a_1,\ldots,a_n)) & \text{caso contrário} \end{cases}$$

- Dado um estado inicial S_0, a execução repetida de R é uma seqüência de estados S_0, S_1, S_2, ..., onde S_{i+1} é o estado obtido pelo disparo de R em S_i.

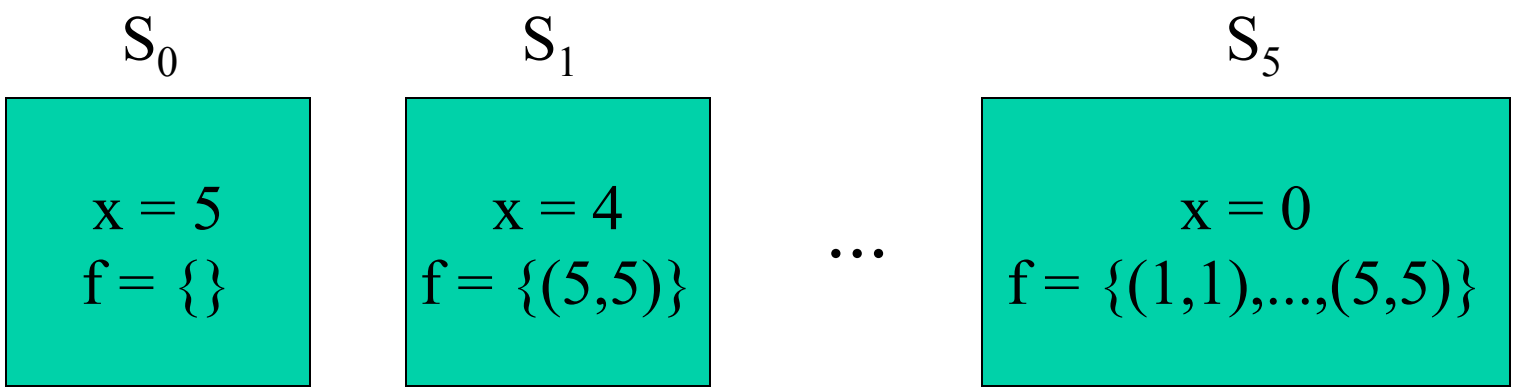
Observação

- O fluxo de controle é diferente do modelo tradicional de linguagens imperativas:
  - O bloco "$x := y, y := x$" tem o efeito de trocar os valores de x e y.
  - Não há construções de repetição semelhantes a um *while*; o comando "**while** *E* **do** *C* **end**" é simulado em ASM por uma regra condicional da forma "**if** *E* **then** *C* **end**".

Exemplo de Loop

- Definição da função f = {(1,1), (2,2), ..., (k,k)}, onde k é o valor de x no estado inicial.

```
if x > 0 then
      x := x - 1,
      f(x) := x
end
```



Compilação Direta de ASM

- Objetivo: compilar ASM para C.
- Uma regra de transição R é compilada para um loop da forma:

```
INIT: R;
      goto INIT;
```

- Desta maneira, a regra R será executada repetidamente.

Compilação de Regras

- Um bloco de regras é compilado como uma seqüência de comandos de C, na ordem em que aparecem.
- Ex: `x := 1, f(2) := x`
- Atualizações só podem ter efeito no passo seguinte.
- Solução: lista de atualizações.

Compilação de Regras...

- Cada regra de atualização é traduzida para uma inserção na “lista de atualizações”.
- Somente ao fim do passo (antes do `goto`), esta lista é percorrida e todas as atualizações são realmente feitas.
- Desta maneira, toda consulta a uma função no passo retorna o valor anterior.

Exemplo

- Considere a regra de transição:

```
f(x) := 1 + g(k), g(p) := 1 + f(x)
```

- O código gerado será o seguinte:

```
INIT: SAVE_UPDATE(f[x], 1 + g[k]);
      SAVE_UPDATE(g[p], 1 + f[x]);
      FIRE_UPDATES;
      goto INIT;
```

- `SAVE_UPDATE` e `FIRE_UPDATES` são macros para salvar uma atualização na lista e disparar as atualizações da lista respectivamente.

Exemplo: Programa ASM

```
if i < n then
   if k < i then
      f(k) := f(i),
      f(i) := 1 + f(k),
      k := k + 1
   else
      i := i + 1
   end
end
```

Exemplo: Tradução Para C

```
INIT:
  if (i < n) {
     if (k < i) {
        SAVE_UPDATE(f[k], f[i]);
        SAVE_UPDATE(f[i], 1 + f[k]);
        SAVE_UPDATE(k, k + 1);
     }
     else
        SAVE_UPDATE(i, i + 1);
     FIRE_UPDATES;
     goto INIT;
  }
```

Otimização de Código ASM

- Principais Causas de Ineficiência:
  - Lista de atualizações vs. atualização direta.
  - Regras guardadas são inerentes ao modelo.
- Principais Fontes de Otimização:
  - Escalonamento de Código
  - Otimização de Desvios
  - Implementação Eficiente de Listas de Atualização

Escalonamento de Código

- A compilação gera código na ordem em que aparece. Exemplo:

```
x := 1, f(2) := x
```

- Outra ordenação pode produzir código de melhor qualidade. Exemplo:

```
f(2) := x, x := 1
```

- Maximizar atualização direta.
- Minimizar o uso de lista de atualizações.

Escalonamento de Código...

- Objetivo: encontrar uma ordem para as instruções na qual o número de inserções necessárias na lista de atualizações seja mínimo.
- A ordem escolhida deve garantir que toda consulta a um identificador retorne o valor do passo anterior e não o que foi atribuído no passo atual.

Algoritmo de Escalonamento

- O escalonamento é feito para cada bloco em três fases:
  - coleta das informações de modificações e consultas do bloco
  - construção do grafo de conflitos
  - escalonamento das instruções do bloco

Fase 1. Coleta das Informações

- Determina quais funções são modificadas e quais são consultadas dentro do bloco.
- Exemplo:

		MODIFICA	CONSULTA
1	`f(x) := 1 + g(k)`	`f`	`x g k`
2	`f(y) := x-k*f(1)`	`f`	`y x k f`
3	`a := f(1) + f(2)`	`a`	`f`
4	`x := g(y)`	`x`	`g y`
5	`y := f(x)`	`y`	`f x`

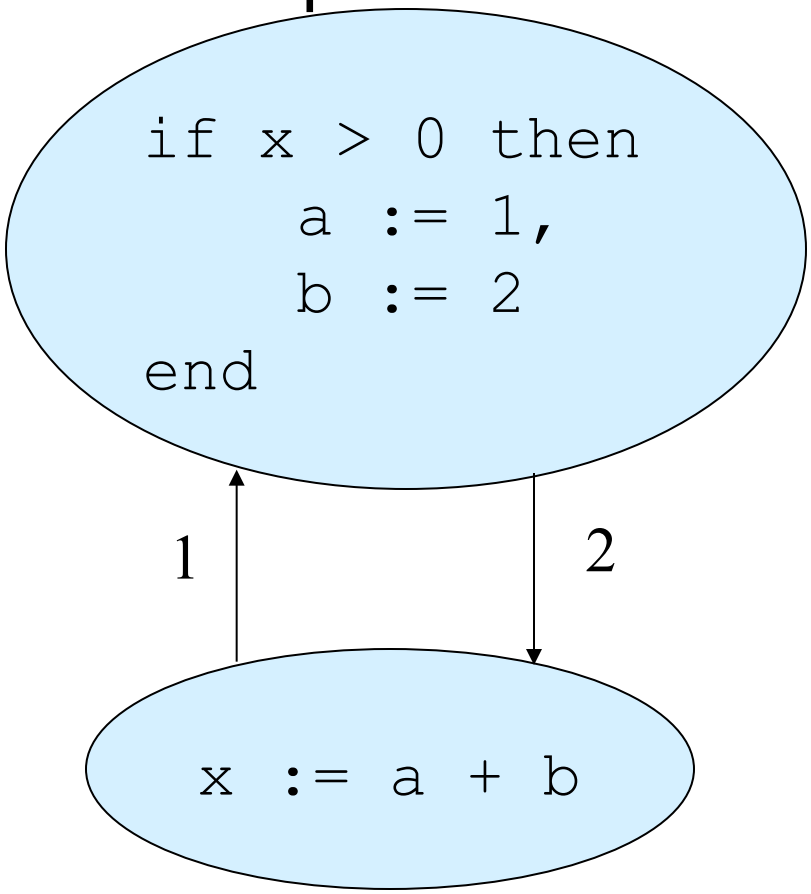
Fase 2. Grafo de Conflitos

- O grafo de conflitos de um bloco é formado da seguinte maneira:
  - Os vértices são os componentes do bloco;
  - Existe um arco unindo R_1 a R_2 se R_1 modificar alguma variável consultada por R_2.
  - A cada arco está associado um peso, que representa o número de conflitos, isto é, o número de atualizações de R_1 cujo alvo é consultado em R_2.

Peso de um Arco

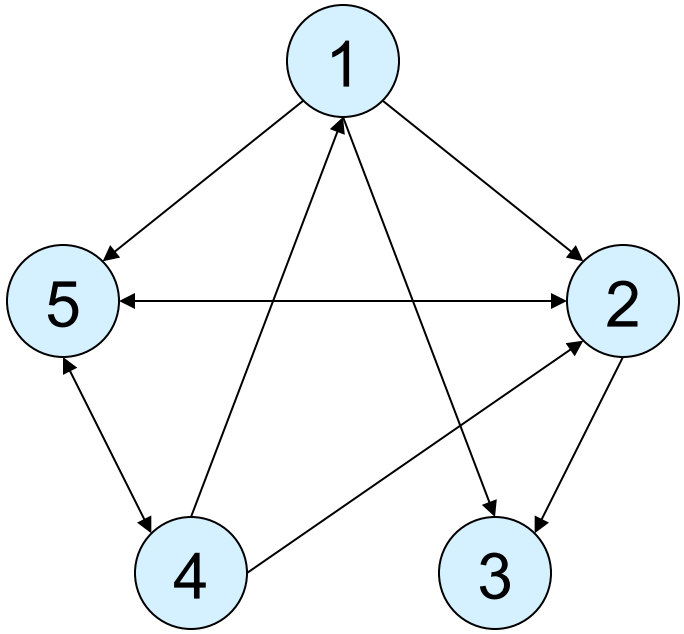
- Número de conflitos entre os componentes do bloco.
- Exemplo:

```
if x > 0 then
    a := 1,
    b := 2
end,
x := a + b
```



Fase 2. Grafo de Conflitos...

	M	C
1	f	x g k
2	f	y x k f
3	a	f
4	x	g y
5	y	f x



Observação: Todos os pesos são iguais a 1

Fase 3. Escalonamento

- Encontrar uma ordem na qual evitem-se, o máximo possível, inserções na lista de atualizações.
- Problema NP-Completo: solução heurística.
- Heurística:
  - retirar vértices do grafo até que o grafo esteja vazio;
  - a condição satisfeita para a escolha do vértice determina se haverá inserções na lista.

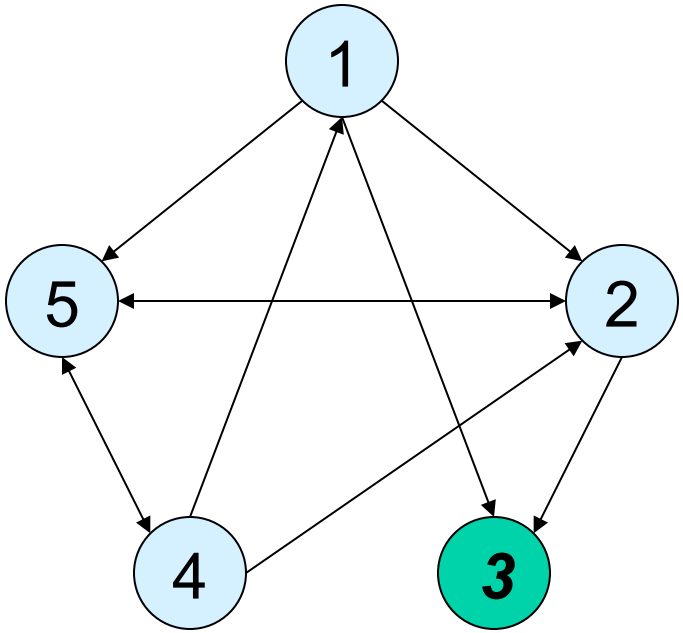
Fase 3. Escalonamento...

- Condições:
  - Para cada vértice v cujo grau de saída seja igual a zero gera-se atualização direta e remove-se do v grafo.
  - Se o caso anterior não se aplicar, remove-se um vértice v tal que a soma dos pesos dos arcos que chegam a v seja o máximo. Insere-se na lista.
  - Em caso de empate, o vértice removido deve ser o que tenha o menor grau de saída. Insere-se na lista.

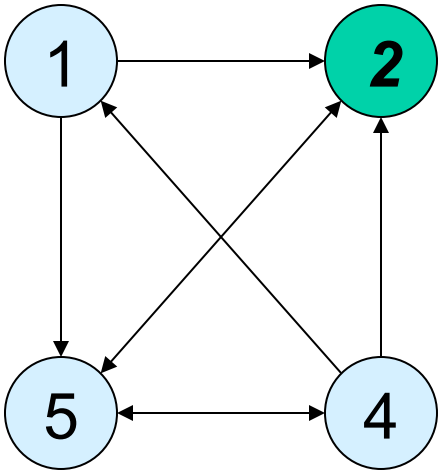
Nenhuma função atualizada em v é consultada nos demais componentes.

Isto diminuirá o grau de saída do maior número de vértices, o que pode levar a mais ocorrências do primeiro caso.

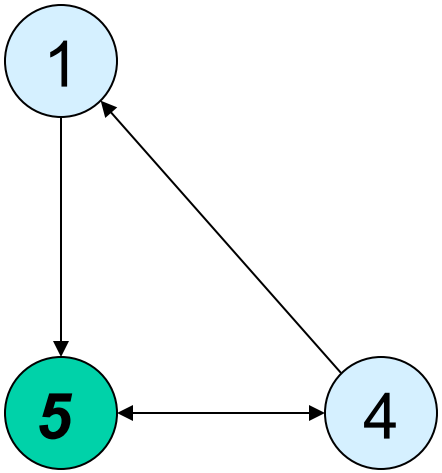
Fase 3. Escalonamento...



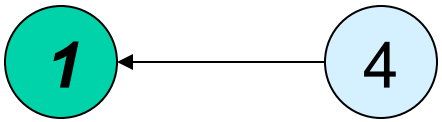
(C1)



(C3)



(C2)



(C1)

(C1)

Fase 3. Escalonamento...

	Inicial		Escalonado
1	`f(x) := 1 + g(k)`	3	`a := f(1) + f(2)`
2	`f(y) := x-k*f(1)`	2	`f(y) := x-k*f(1)`
3	`a := f(1) + f(2)`	5	`y := f(x)`
4	`x := g(y)`	1	`f(x) := 1 + g(k)`
5	`y := f(x)`	4	`x := g(y)`

OBS: As atualizações 2 e 5 são inseridas na lista de atualizações, pois os alvos das atualizações são utilizados em instruções seguintes.

Otimização de Desvios

- Programas escritos em ASM são baseados em regras guardadas e aninhadas:

```
if g1 then
    if g2 then
        if g3 then
            ... R1 ...
    else
        if g4 then
            ... R2 ...
end
```

Otimização de Desvios...

- Objetivo: determinar o melhor ponto onde começar o próximo passo, procurando por condições que não foram modificadas no passo atual.

Otimização de Desvios...

- Exemplo:

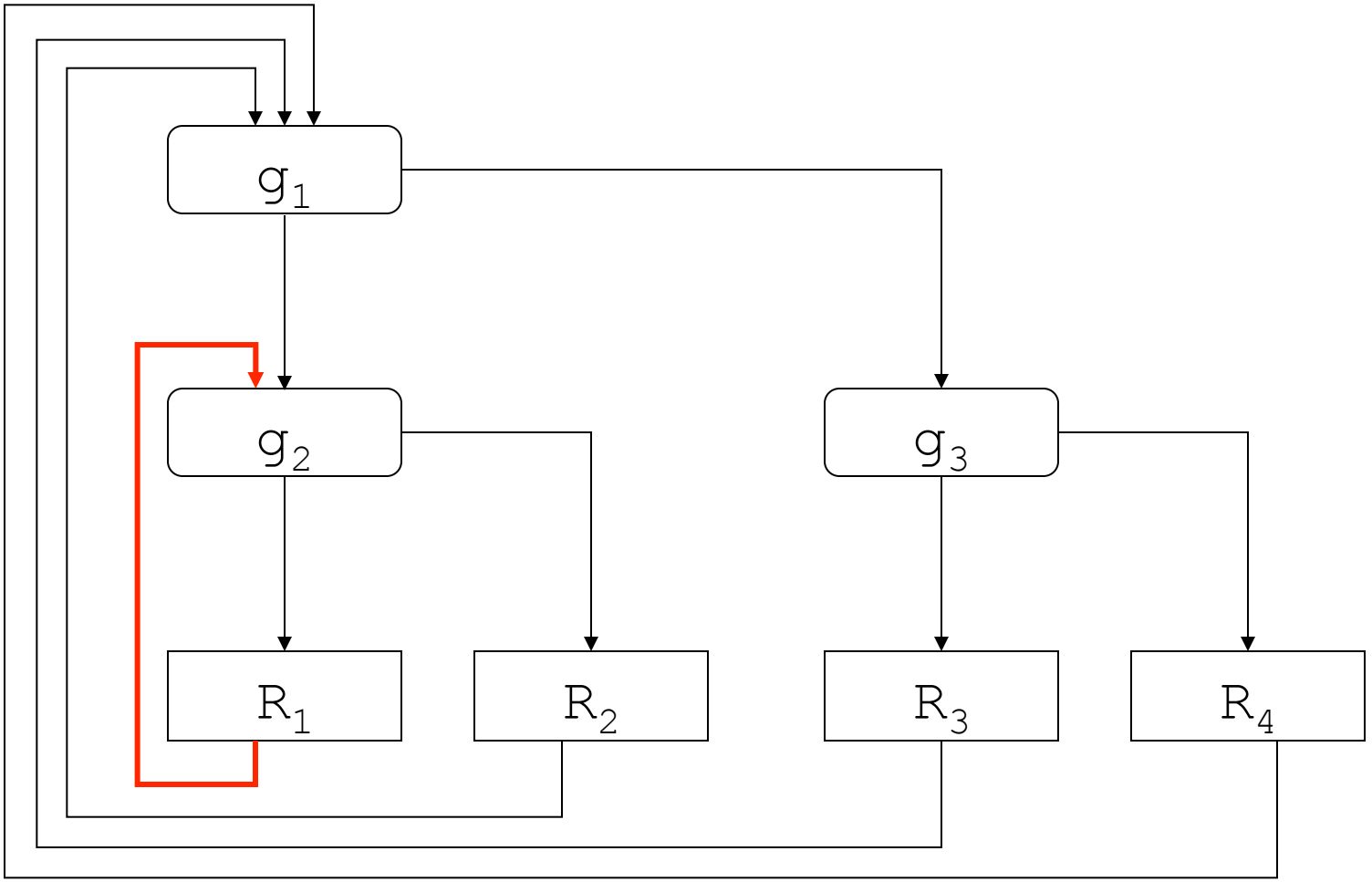
```
if g1 then
   if g2 then R1 else R2 end
else
   if g3 then R3 else R4 end
end
```

- Supondo que `R1` modifique funções consultadas somente por `g2`, não é necessário retornar diretamente para a avaliação de `g1`.

Otimização de Desvios...

- Associam-se a cada bloco as guardas avaliadas para chegar ao bloco.
- A guarda mais distante que consultar alguma função alterada no bloco é o alvo do desvio para o retorno.

Otimização de Desvios...



Otimização de Desvios...

- Código gerado:

```
EvalG1: if (!g1) goto EvalG3;
EvalG2: if (!g2) goto ExecR2;
ExecR1: ... /* Execução de R1 */
        goto EvalG2;
ExecR2: ... /* Execução de R2 */
        goto EvalG1;
EvalG3: if (!g3) goto ExecR4;
ExecR3: ... /* Execução de R3 */
        goto EvalG1;
ExecR4: ... /* Execução de R4 */
        goto EvalG1;
```

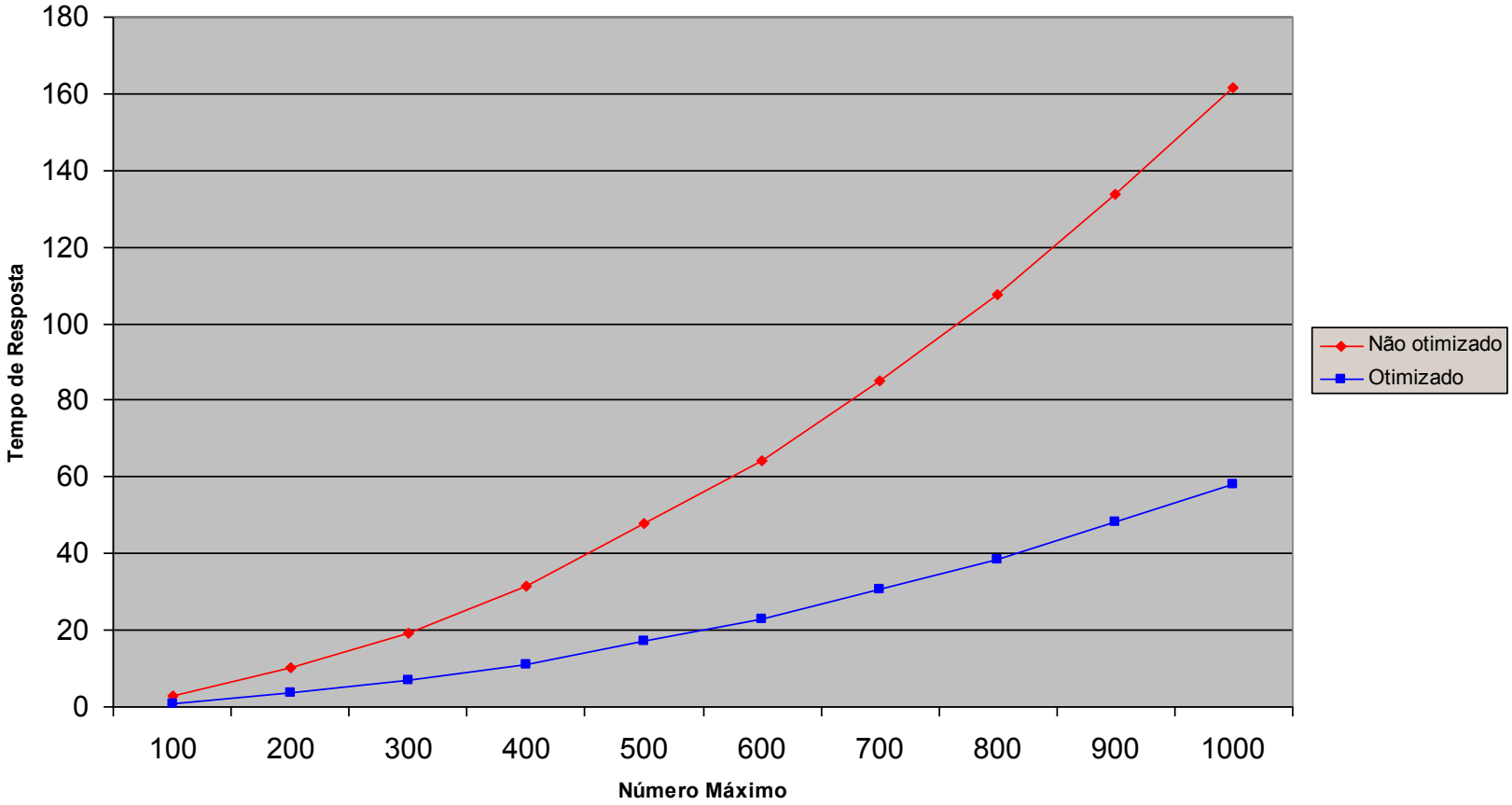
Implementação de Listas de Atualizações

- Os nodos de uma lista de atualização são pares da forma *(L, R)*, onde:
  - L é o endereço da atualização
  - R é o valor atribuído (de qualquer tipo)
- O polimorfismo da lista de atualizações deve ser evitado por razões de eficiência.
- Solução: alocação espelhada.

Avaliação das Técnicas Propostas

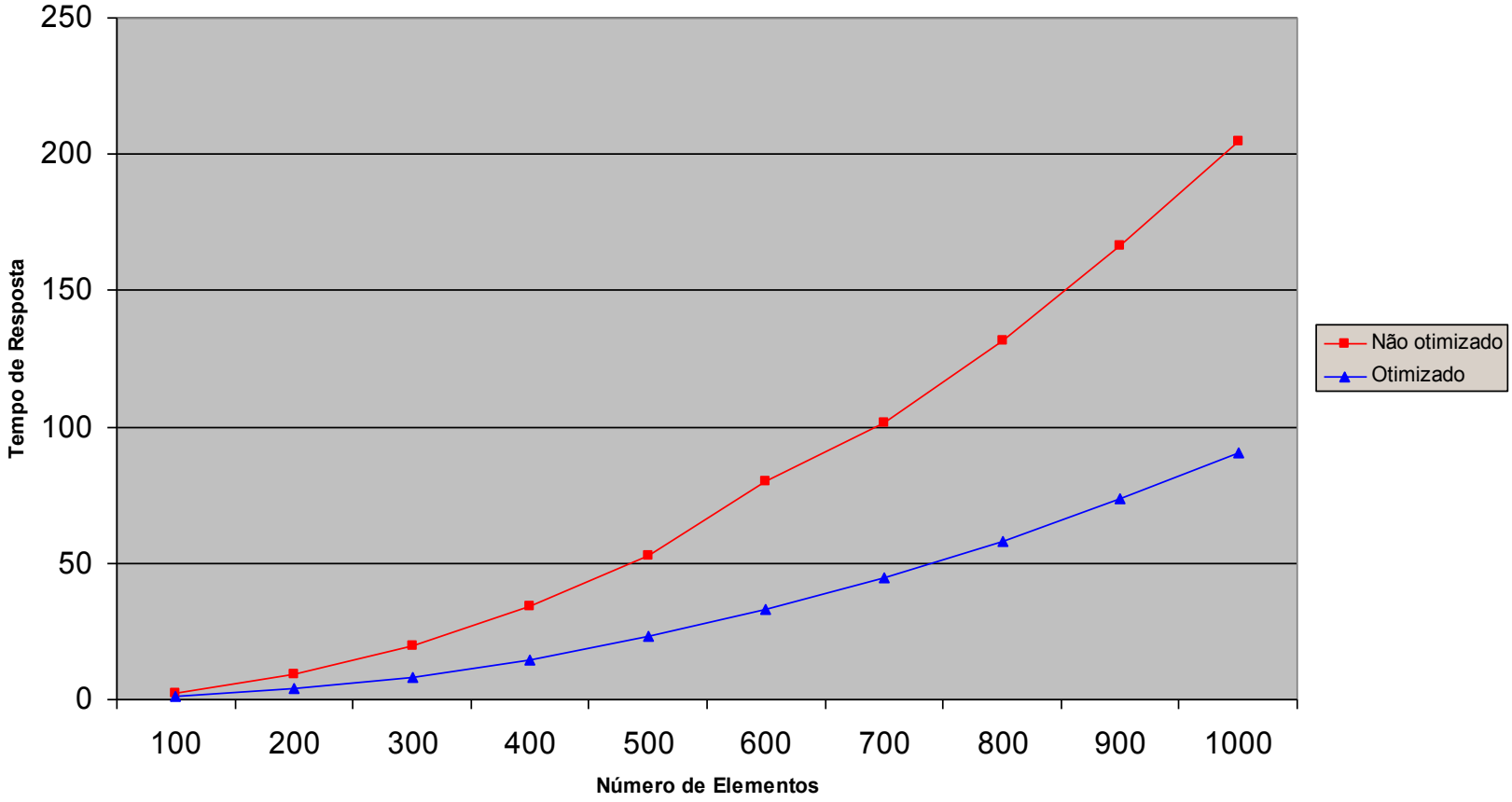
- *Benchmarks* utilizados:
  - Determinação de primos.
  - Algoritmo de ordenação por seleção.

Primos - Listas de Atualização



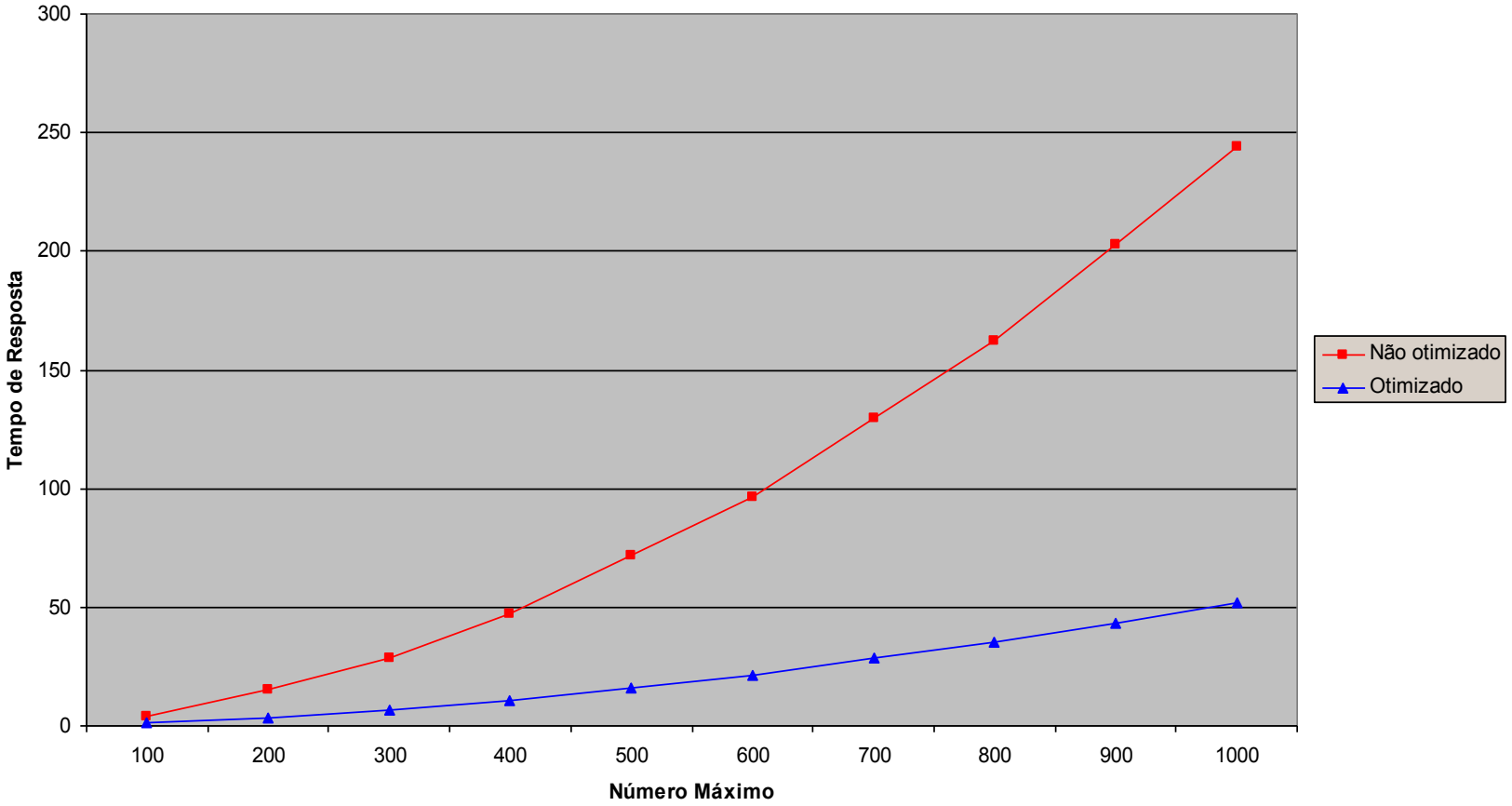
Melhoria = 2,86 x mais rápido

Ordenação - Listas de Atualização



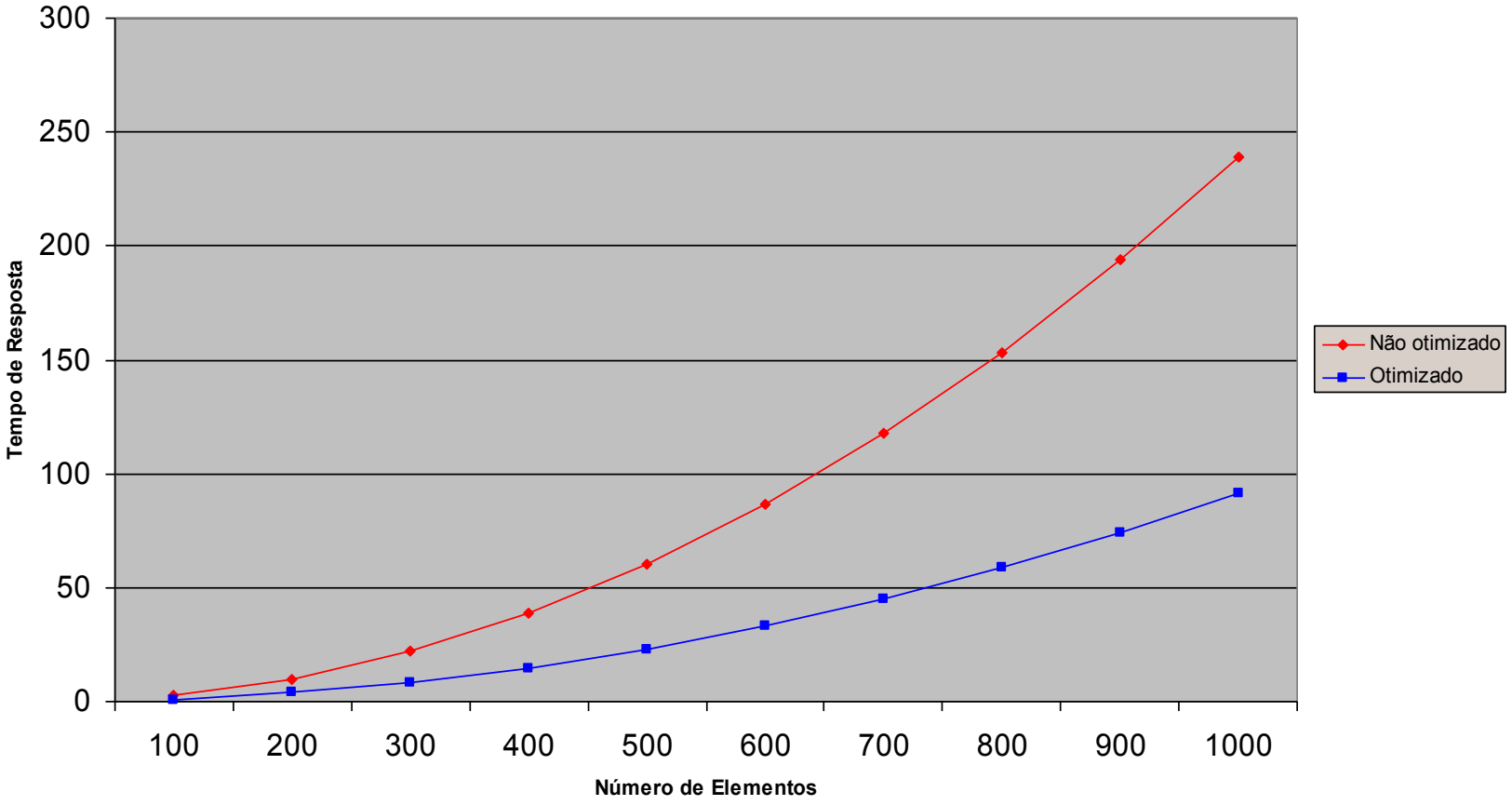
Melhoria = 2,33 x mais rápido

Primos - Escalonamento



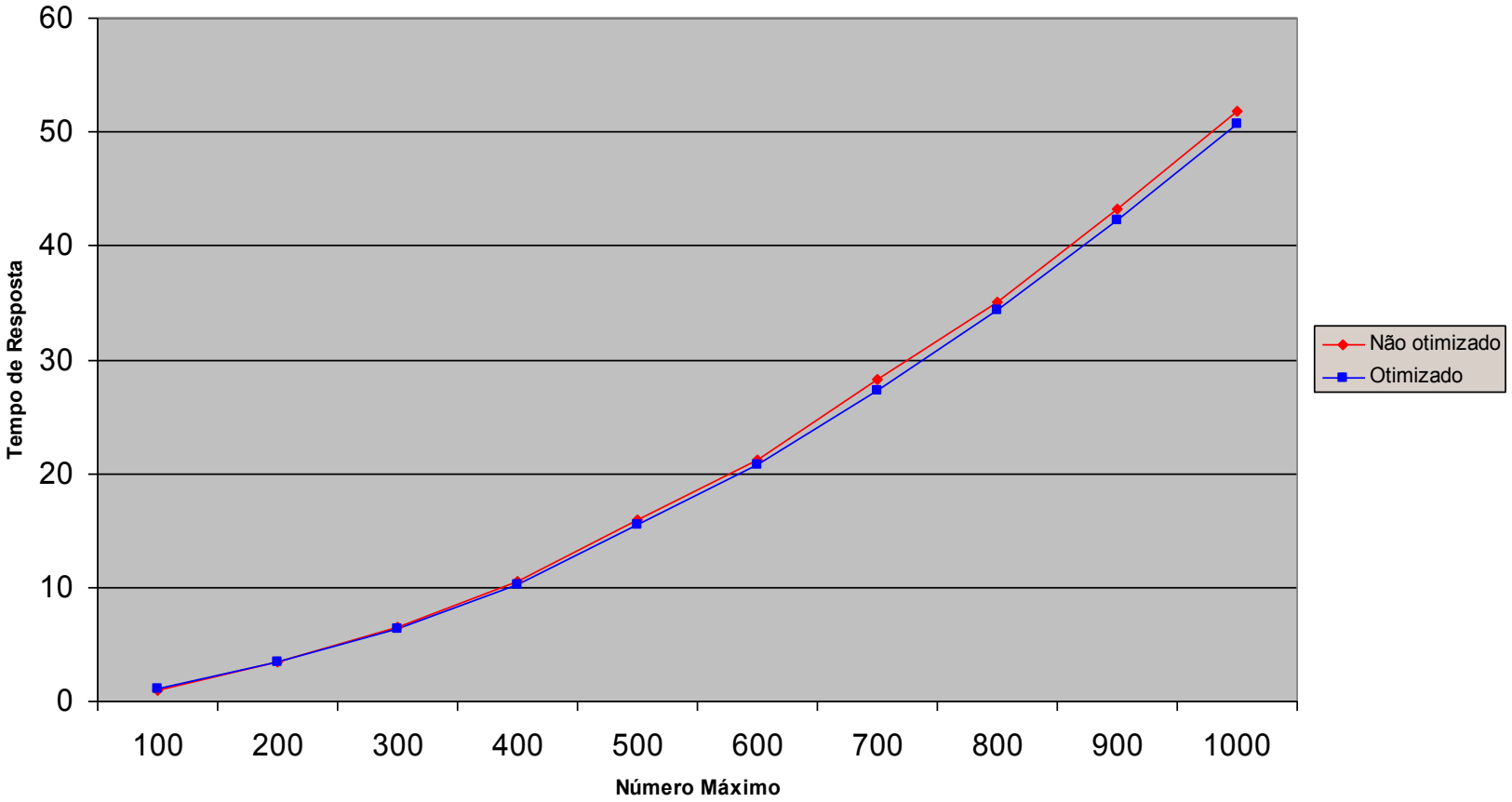
Melhoria = 4,55 x mais rápido

Ordenação - Escalonamento



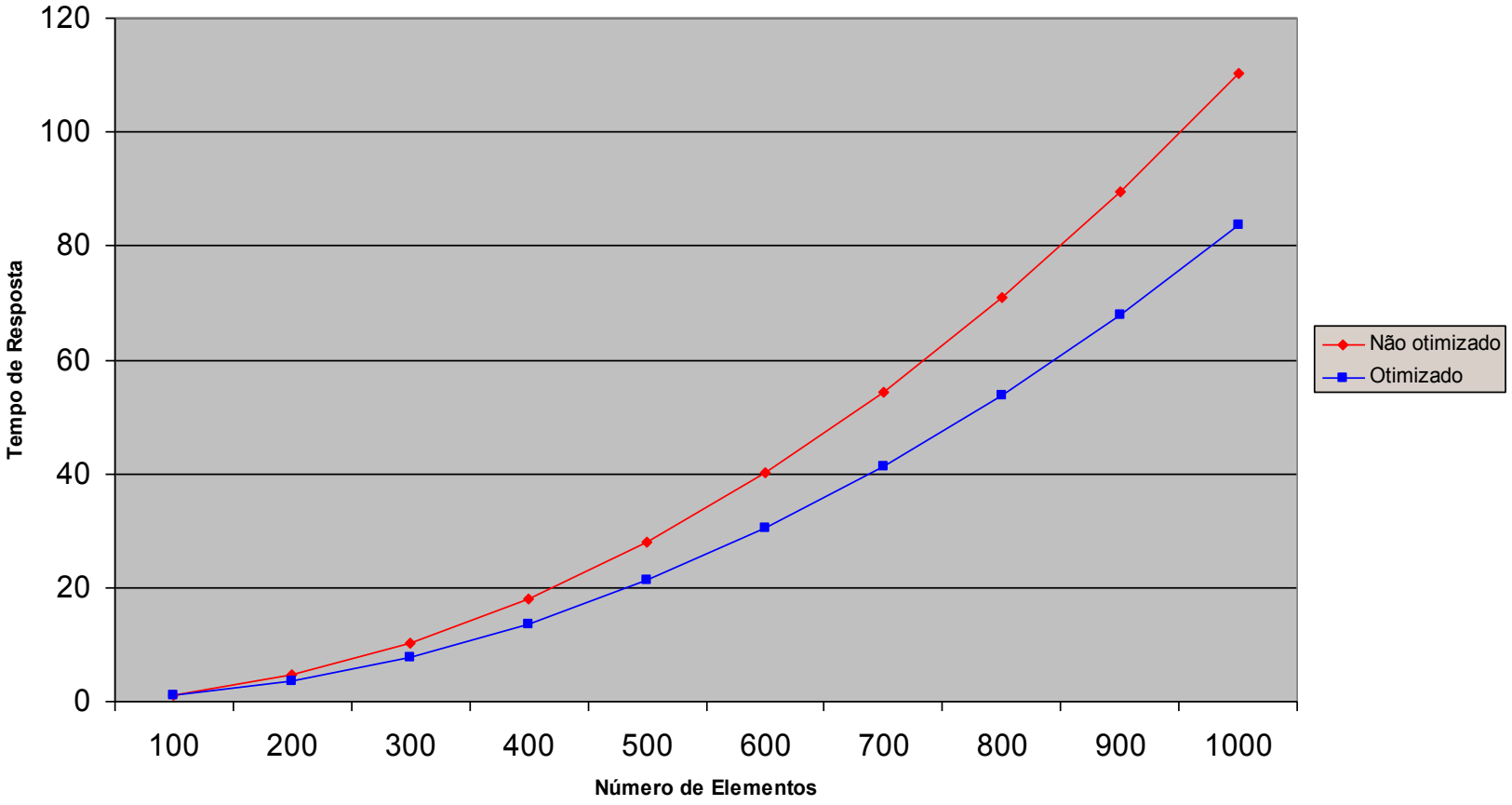
Melhoria = 2,63 x mais rápido

Primos - Otimização de Desvios



Melhoria = 1,01 x mais rápido

Ordenação - Otimização de Desvios



Melhoria = 1,30 x mais rápido

Avaliação das Técnicas Propostas...

- Utilizando as técnicas nos programas acima:

  (tempo otimizado/tempo não-otimizado)

  Primos:

  35% (L) 22% (E) 99% (D) = 16,6 x mais rápido

  Ordenação:

  43% (L) 38% (E) 77% (D) = 7,9 x mais rápido

  L = Listas de atualizações com alocação espelhada,
  E = Escalonamento
  D = Desvios

Ordenação Por Seleção
(Estado Inicial)

```
modo = 1
i = 1
n = Tamanho do array
```

Ordenação Por Seleção
(Regra de Transição)

```
if modo = 1 and i < n then
   k := i, j := i + 1, modo := 2
elseif modo = 2 then
   if j > n then
      modo := 3
   else
      if array(j) < array(k) then
         k := j
      end,
      j := j + 1
   end
elseif modo = 3 then
   if k != i then
      array(k) := array(i), array(i) := array(k)
   end,
   i := i + 1, modo := 1
end
```

Números Primos
(Estado Inicial)

```
phase = 0
x = 2
primes(x) = true, for all x in {2,n}
```

Números Primos
(Regra de Transição)

```
if phase = 0 and x <= n then
   if y < x then
      if primes(y) and x % y = 0 then
         primes(x) := false
         phase := 1
      end,
      y := y + 1
   else
      phase := 1
   end
else if phase = 1 then
   x := x + 1
   phase := 0
end
```