Studying Optimal Spilling in the light of SSA

Quentin Colombet, Florian Brandner and Alain Darte
Compsys, LIP, UMR 5668 CNRS, INRIA, ENS-Lyon, UCB-Lyon

Journées compilation, Rennes, France, June 18-20 2012
Outline

1. Introduction
2. Formulating “Optimal” Spilling
3. A “More Optimal” Formulation
4. Experiments
5. Conclusion
Outline

1. Introduction
2. Formulating “Optimal” Spilling
3. A “More Optimal” Formulation
4. Experiments
5. Conclusion
Introduction

Register Allocation

Map an unlimited number of virtual variables to actual physical registers.
Register Allocation
Map an unlimited number of virtual variables to actual physical registers.

- Simplified graph coloring based approach
Register Allocation

Map an unlimited number of virtual variables to actual physical registers.

- Simplified graph coloring based approach

- Build: Build the interference graph (IG)
- Simplify: Apply Kempe’s Algorithm
- Spill: Evict one variable into memory (spill-everywhere)
- Coloring: Assign color using order from simplify
Introduction

Register Allocation

Map an unlimited number of virtual variables to actual physical registers.

- Simplified graph coloring based approach
  - Build: Build the interference graph (IG)
  - Simplify: Apply Kempe’s Algorithm
  - Spill: Evict one variable into memory (spill-everywhere)
  - Coloring: Assign color using order from simplify

- Decoupled approach
  - Spill: #live variables ≤ K for each program point
  - Coloring: Assign variables colors
Register Allocation

Map an unlimited number of virtual variables to actual physical registers.

- Simplified graph coloring based approach
  
  Build: Build the interference graph (IG)
  Simplify: Apply Kempe’s Algorithm
  Spill: Evict one variable into memory (spill-everywhere)
  Coloring: Assign color using order from simplify

- Decoupled approach
  
  Spill: \#live variables ≤ K for each program point
  Coloring: Assign variables colors
## Problematic

### Context
- Decoupled register allocation
  - Spill
  - Assignment
- Based on static single assignment (SSA)
Problematic

Context
- Decoupled register allocation
  - Spill
  - Assignment
- Based on static single assignment (SSA)

Motivations
- Study impact of SSA on spilling
  - Chordal interference graphs help for assignment
  - Does SSA help for spilling too?
- Evaluate existing spilling heuristic
Problematic

Context
- Decoupled register allocation
  - Spill
  - Assignment
- Based on static single assignment (SSA)

Motivations
- Study impact of SSA on spilling
  - Chordal interference graphs help for assignment
  - Does SSA help for spilling too?
- Evaluate existing spilling heuristic

Contributions
- Provide an exact formulation
- Exploit variable-to-variable copies
- Discuss existing spilling models
Static Single Assignment (SSA)

SSA provides sufficient split points, unless pre-coloring or aliasing is involved.
Static Single Assignment (SSA)

SSA provides sufficient split points, unless pre-coloring or aliasing is involved.

\[
\begin{align*}
    &a = \\
    &c = \\
    &b_1 = a \\
    &\quad = c \\
    &b_2 = \\
    &\quad = b_2, a \\
    &b_3 = \phi(b_1, b_2) \\
    &\quad \doteq b_3
\end{align*}
\]
Static Single Assignment (SSA)

SSA provides sufficient split points, unless pre-coloring or aliasing is involved.

\[
a = \phi(b_1, b_2) = b_3
\]

Properties

Every use has at most one reaching definition

For strict SSA: A definition dominates all its uses.

I.e. it does not exist a path from the function entry to \( v \)'s use that does not traverse \( v \)'s definition.
Static Single Assignment (SSA)

SSA provides sufficient split points, unless pre-coloring or aliasing is involved.

\[
\begin{align*}
a &= \\
c &= \\
b_1 &= a \\
&= c \\
b_2 &= \\
&= b_2, a \\
b_3 &= \phi(b_1, b_2) \\
&= b_3
\end{align*}
\]
Static Single Assignment (SSA)

SSA provides sufficient split points, unless pre-coloring or aliasing is involved.

\[
\begin{align*}
\text{a} &= c = b_1 = a = c \\
\text{b}_2 &= b_2, a \\
\text{b}_3 &= \phi(b_1, b_2) = b_3
\end{align*}
\]

Properties

- Every use has at most one reaching definition
- For strict SSA: A definition dominates all its uses.
  I.e. it does not exist a path from the function entry to \( v \)'s use that does not traverse \( v \)'s definition.
Outline

1 Introduction

2 Formulating “Optimal” Spilling

3 A “More Optimal” Formulation

4 Experiments

5 Conclusion
Existing Approaches

Various 'optimal' approaches:

- Integer Linear Programming (ILP)
  - Appel & George
  - related Goodwin & Wilken (ORA)

- Multi-Commodity Network Flow
  - Koes & Goldstein

- Constrained Min-Cut
  - Ebner & Scholz & Krall
Existing Approaches

Various ’optimal’ approaches:

- Integer Linear Programming (ILP)
  - Appel & George
  - related Goodwin & Wilken (ORA)
- Multi-Commodity Network Flow
  - Koes & Goldstein
- Constrained Min-Cut
  - Ebner & Scholz & Krall
- In the end these approaches rely on ILP.
Existing Approaches

Various ‘optimal’ approaches:

- Integer Linear Programming (ILP)
  - Appel & George
  - related Goodwin & Wilken (ORA)

- Multi-Commodity Network Flow
  - Koes & Goldstein

- Constrained Min-Cut
  - Ebner & Scholz & Krall

- In the end these approaches rely on ILP.

All of these formulations have surprising flaws!
Flaws: Liveness\(^1\)

\[
\begin{align*}
  a &= \ldots \\
  b &= a + 1 \quad \checkmark \\
  \ldots &= a \\
  (a) \text{ before spilling}
\end{align*}
\]

\[
\begin{align*}
  a &= \ldots \\
  b &= a + 1 \quad \checkmark \\
  \ldots &= a \\
  (b) \text{ ineffective spilling}
\end{align*}
\]

Problem:
- Variables are either available in memory or register (exclusive)
- Load/store required to change availability
- Artificial interference between \(a\) and \(b\)

\(^1\) Applies to: Appel, Koes; in other form also Goodwin
Flaws: Spurious Spill Code

a = ...
while(...){
  if (...)
    store a
  else
    load a
      = a
  }

(a) Koes 1

\(^2\text{Applies to: Appel, Koes}\)
Flaws: Spurious Spill Code\(^2\)

(a) Koes 1

\[
\begin{align*}
a &= \ldots \\
\text{while}(\ldots)&\{
\quad\text{if (\ldots)} \\
\quad\quad\text{store } a \\
\quad\quad\text{\(\not\)
load } a \\
\quad\quad\quad= a \\
\quad\text{else} \\
\quad\quad\text{\(\not\)
load } a \\
\quad\quad\quad= a \\
\}\}
\]

(b) Koes 2

\[
\begin{align*}
a &= \ldots \\
\text{store } a \\
\text{while}(\ldots)&\{
\quad\text{if (\ldots)} \\
\quad\quad\text{\(\not\)
load } a \\
\quad\quad\quad= a \\
\quad\text{else} \\
\quad\quad\text{\(\not\)
load } a \\
\quad\quad\quad= a \\
\}\}
\]

\[
\begin{align*}
\quad\text{store } a
\end{align*}
\]

(c) Koes 3

\[
\begin{align*}
a &= \ldots \\
\text{store } a \\
\text{load } a \\
\text{while}(\ldots)&\{
\quad\text{if (\ldots)} \\
\quad\quad\text{\(\not\)
load } a \\
\quad\quad\quad= a \\
\quad\text{else} \\
\quad\quad\text{\(\not\)
load } a \\
\quad\quad\quad= a \\
\}\}
\]

\[
\begin{align*}
\quad\text{store } a
\end{align*}
\]

\(^2\)Applies to: Appel, Koes
Flaws: Spurious Spill Code

\[
\begin{align*}
\text{a} &= \ldots \\
\text{while}(...) &\{ \\
\quad \text{if} (...) &\{ \\
\qquad \text{store a} \\
\qquad \text{load a} &\{ \\
\qquad\quad = \text{a} \\
\qquad\quad = \text{a} \\
\quad \text{else} &\{ \\
\quad\quad = \text{a} \\
\quad\quad = \text{a} \\
\quad \} \\
\} \\
\} \\
\text{(a) Koes 1}
\end{align*}
\]

\[
\begin{align*}
\text{a} &= \ldots \\
\text{store a} \\
\text{while}(...) &\{ \\
\quad \text{if} (...) &\{ \\
\qquad \text{load a} &\{ \\
\qquad\quad = \text{a} \\
\quad\quad = \text{a} \\
\quad \text{else} &\{ \\
\quad\quad = \text{a} \\
\quad\quad = \text{a} \\
\quad \} \\
\} \\
\} \\
\} \\
\text{(b) Koes 2}
\end{align*}
\]

\[
\begin{align*}
\text{a} &= \ldots \\
\text{store a} \\
\text{load a} \\
\text{while}(...) &\{ \\
\quad \text{if} (...) &\{ \\
\qquad \text{store a} \\
\qquad \text{load a} &\{ \\
\qquad\quad = \text{a} \\
\quad\quad = \text{a} \\
\quad \text{else} &\{ \\
\quad\quad = \text{a} \\
\quad\quad = \text{a} \\
\quad \} \\
\} \\
\} \\
\} \\
\text{(c) Koes 3}
\end{align*}
\]

\[\text{Applies to: Appel, Koes}\]
Limitations

Limitations of existing approaches:

- **Rematerialization**
  - Appel & George: None
  - related Goodwin & Wilken: Simple and partial
  - Koes & Goldstein: Simple and partial
  - Ebner & Scholz & Krall: None

- **Support of SSA**
  - Ebner & Scholz & Krall: Partial
  - Others: None
Limitations

Limitations of existing approaches:

- **Rematerialization**
  - Appel & George: None
  - related Goodwin & Wilken: Simple and partial
  - Koes & Goldstein: Simple and partial
  - Ebner & Scholz & Krall: None

- **Support of SSA**
  - Ebner & Scholz & Krall: Partial
  - Others: None

*We design a new model*
Outline

1. Introduction

2. Formulating “Optimal” Spilling

3. A “More Optimal” Formulation

4. Experiments

5. Conclusion
Express spilling using ILP:

- Availability around program points
  - Available in memory / in register
  - Non-exclusive!!

- Actions on program points
  - Load, store, rematerialization, ...

- Propagation along points
  - Along edges in the control flow graph (CFG)
  - Between operations within basic blocks accounting for uses/definitions
Our Formulation - Features

**Basic**
- Load/Store placement
- Simple rematerialization

This model can emulate all existing approaches.
Our Formulation - Features

Basic
- Load/Store placement
- Simple rematerialization

This model can emulate all existing approaches.

Extended
- Features of basic model
- Copy/SSA handling
- Generalized rematerialization

The extended model is able to emulate the basic one.
SSA Specificities

$\phi$-Operations represent implicit copies:

\[
\begin{align*}
    a &= \phi(b, c) \\
    e &= \phi(b, d)
\end{align*}
\]

(a) SSA form
SSA Specificities

ϕ-Operations represent implicit copies:

\[
\begin{align*}
\downarrow & \quad \uparrow \\
(a, e) &= (a_{\phi}, e_{\phi}) \\
(a, e) &= (a_{\phi}, e_{\phi})
\end{align*}
\]

\[
\begin{align*}
(a\phi, e\phi) &= (b, b) & (a\phi, e\phi) &= (c, d) \\
(a, e) &= (a\phi, e\phi)
\end{align*}
\]

(a) SSA form

(b) Transform ϕ-operations

Simple approach: spilling as if not under SSA form.
SSA Specificities

\( \phi \)-Operations represent implicit copies:

\[
\begin{align*}
&\downarrow \quad \downarrow \\
&a = \phi(b, c) \quad (a_\phi, e_\phi) = (b, b) \quad (a_\phi, e_\phi) = (c, d) \\
&e = \phi(b, d) \\
&\quad \downarrow \quad \downarrow \\
&(a, e) = (a_\phi, e_\phi)
\end{align*}
\]

(a) SSA form
(b) Transform \( \phi \)-operations

Simple approach: spilling as if not under SSA form.

Example: Appel & George with and without SSA:

- Spill cost:
  - 5% worse on average under SSA
  - Best improvement: 2%
  - Worst case: 50%
**SSA Specificities**

\( \phi \)-Operations represent implicit copies:

\[
\begin{align*}
    a &= \phi(b, c) \\
    e &= \phi(b, d) \\
    (a, e) &= (a, e)
\end{align*}
\]

(a) SSA form

\[
\begin{align*}
    (a_\phi, e_\phi) &= (b, b) \\
    (a_\phi, e_\phi) &= (c, d)
\end{align*}
\]

(b) Transform \( \phi \)-operations

Simple approach: spilling as if not under SSA form.

Example: Appel & George with and without SSA:

- Spill cost:
  - 5% worse on average under SSA
  - Best improvement: 2%
  - Worst case: 50%

⇒ Copies force variables to be in register.
Handling $\phi$ and Copy Operations

- **Basic:**
  - Replace $\phi$-operations by copies
  - Sequentialize Copies
  - Treat copies as normal operations
Handling $\phi$ and Copy Operations

- **Basic:**
  - Replace $\phi$-operations by copies
  - Sequentialize Copies
  - Treat copies as normal operations

- **Optimistic:**
  - Copies/$\phi$s are virtual operations
  - Propagate locations through them
  - Coalesce memory slots afterwards
  - Repair when memory slots cannot be shared
Handling $\phi$ and Copy Operations

- **Basic:**
  - Replace $\phi$-operations by copies
  - Sequentialize Copies
  - Treat copies as normal operations

- **Optimistic:**
  - Copies/$\phi$s are virtual operations
  - Propagate locations through them
  - Coalesce memory slots afterwards
  - Repair when memory slots cannot be shared

- **Pessimistic:**
  - Replace $\phi$-operations by copies
  - Copies are virtual operations
  - Propagate locations through a subset of copies
  - Coalesce remaining memory slots afterwards
Outline

1 Introduction

2 Formulating “Optimal” Spilling

3 A “More Optimal” Formulation

4 Experiments

5 Conclusion
Experiments

Setup:

- Production compiler for STMicroelectronics ST2xx VLIW
  - 4-way parallel
  - 32KB direct mapped I-cache
  - 32KB 4-way set associative D-cache
  - 1 load/store per cycle
  - 3 cycles load-use delay
- Restricted to 8 registers
- SPEC2000 and EEMBC v1.1 benchmarks
- IBM CPLEX 12.2 with 1000s time limit
Experiments (2)

Configurations:

- **Appel-G.** Appel and George’s ILP Formulation
- **Coloring** Heuristic using iterated register coalescing
- **SpEv** Basic formulation emulating spill everywhere
- **Basic** Our basic formulation
- **BasicSSA** Naive handling of SSA
- **Pessimistic** Extended formulation, pessimistic coalescing sets
- **Optimistic** Extended formulation, optimistic coalescing sets
- **SpEv_ssa** Emulation of spill everywhere under SSA
- **Hack** Hack’s SSA-based spilling heuristic
Spill Costs (EEMBC)

(Lower is better)
Runtime (EEMBC)
Spill Costs (SPEC)

(Lower is better)
Outline

1. Introduction
2. Formulating “Optimal” Spilling
3. A “More Optimal” Formulation
4. Experiments
5. Conclusion
Conclusion

- Accurate ILP formulation for spilling
  - Copy-relations and coalescing
  - Emulation of other approaches
Conclusion

- Accurate ILP formulation for spilling
  - Copy-relations and coalescing
  - Emulation of other approaches
- SSA form complicates matters
  - Parallel $\phi$-semantics and memory coalescing
  - Ignoring $\phi$s gives unpredictable behavior
Conclusion

- Accurate ILP formulation for spilling
  - Copy-relations and coalescing
  - Emulation of other approaches
- SSA form complicates matters
  - Parallel $\phi$-semantics and memory coalescing
  - Ignoring $\phi$s gives unpredictable behavior
- Placement of spill code is important
  - Spill costs alone are a bad metric
  - State of pipeline and memory subsystem have to be considered
Handling $\phi$ and Copy Operations - Strategies

Copy related variables can share a memory slot:

- **Pessimistic**: If they do not interfere in the original program
Handling $\phi$ and Copy Operations - Strategies

Copy related variables can share a memory slot:

- **Pessimistic**: If they do not interfere in the original program

\[
(a_{\phi}, e_{\phi}) = (b, b) \quad (a_{\phi}, e_{\phi}) = (c, d)
\]

\[
(a, e) = (a_{\phi}, e_{\phi})
\]

(a) **Transform $\phi$-operations**
Handling $\phi$ and Copy Operations - Strategies

Copy related variables can share a memory slot:

- **Pessimistic**: If they do not interfere in the original program

\[
(a_\phi, e_\phi) = (b, b) \quad (a_\phi, e_\phi) = (c, d)
\]

\[
(a, e) = (a_\phi, e_\phi)
\]

(a) **Transform $\phi$-operations**

\[
\{a, a_\phi, c\} \quad \{e, e_\phi, d\} \quad \{b\}
\]

(b) **Build coalescing classes**
Handling $\phi$ and Copy Operations - Strategies

Copy related variables can share a memory slot:

- **Pessimistic**: If they do not interfere in the original program

\[(a_\phi, e_\phi) = (b, b) \quad (a_\phi, e_\phi) = (c, d)\]

\[\downarrow \quad \downarrow\]

\[(a, e) = (a_\phi, e_\phi)\]

(a) **Transform $\phi$-operations**

\[\{a, a_\phi, c\} \quad \{e, e_\phi, d\} \quad \{b\}\]

(b) **Build coalescing classes**

\[b = ld@_b\]

\[(a_\phi, e_\phi) = (b, b)\]

\[\@_{a_\phi c} = st \ a_\phi\]

\[\@_{e_\phi d} = st \ e_\phi\]

\[\downarrow \quad \downarrow\]

(c) **Spill**
Handling $\phi$ and Copy Operations - Strategies

Copy related variables can share a memory slot:

- **Pessimistic**: If they do not interfere in the original program

\[(a_\phi, e_\phi) = (b, b) \quad (a_\phi, e_\phi) = (c, d)\]

\[\downarrow \quad \downarrow\]

\[(a, e) = (a_\phi, e_\phi)\]

(a) *Transform $\phi$-operations*

\[\{a, a_\phi, c\} \{e, e_\phi, d\} \{b\}\]

(b) *Build coalescing classes*

\[b = ld_{@b}\]

\[(a_\phi, e_\phi) = (b, b)\]

\[\@_{aa_\phi c} = st \ a_\phi\]

\[\@_{ee_\phi d} = st \ e_\phi\]

\[\downarrow \quad \downarrow\]

(c) *Spill*

- **Optimistic**: Always

\[\downarrow \downarrow\]
Handling $\phi$ and Copy Operations - Strategies

Copy related variables can share a memory slot:

- **Pessimistic**: If they do not interfere in the original program

\[(a_\phi, e_\phi) = (b, b) \quad (a_\phi, e_\phi) = (c, d)\]

\[(a, e) = (a_\phi, e_\phi)\]

(a) **Transform $\phi$-operations**

\[\{a, a_\phi, c\} \quad \{e, e_\phi, d\} \quad \{b\}\]

(b) **Build coalescing classes**

\[b = ld@b\]
\[(a_\phi, e_\phi) = (b, b)\]
\[@_{aa_\phi c} = st \quad a_\phi\]
\[@_{ee_\phi d} = st \quad e_\phi\]

(c) **Spill**

- **Optimistic**: Always

\[a = \phi(b, c)\]
\[e = \phi(b, d)\]

(a) **SSA form**
Handling $\phi$ and Copy Operations - Strategies

Copy related variables can share a memory slot:

- **Pessimistic**: If they do not interfere in the original program

  $$(a_\phi, e_\phi) = (b, b) \quad (a_\phi, e_\phi) = (c, d)$$

  $$(a, e) = (a_\phi, e_\phi)$$

  (a) **Transform $\phi$-operations**

  $$\{a, a_\phi, c\} \quad \{e, e_\phi, d\} \quad \{b\}$$

  (b) **Build coalescing classes**

- **Optimistic**: Always

  $$a = \phi(b, c)$$
  $$e = \phi(b, d)$$

  (a) **SSA form**

  $$\@a = \phi(\@b, \@c)$$
  $$\@e = \phi(\@b, \@d)$$

  (b) **Spill**

$$b = ld@b$$

$$(a_\phi, e_\phi) = (b, b)$$

$$\@aa_\phi c = st \ a_\phi$$

$$\@ee_\phi d = st \ e_\phi$$

(c) **Spill**
Handling $\phi$ and Copy Operations - Strategies

Copy related variables can share a memory slot:

- **Pessimistic**: If they do not interfere in the original program

  \[
  (a_{\phi}, e_{\phi}) = (b, b) \quad (a_{\phi}, e_{\phi}) = (c, d)
  \]

  \[
  (a, e) = (a_{\phi}, e_{\phi})
  \]

  (a) **Transform $\phi$-operations**

  \[
  \{a, a_{\phi}, c\} \{e, e_{\phi}, d\} \{b\}
  \]

  (b) **Build coalescing classes**

- **Optimistic**: Always

  \[
  a = \phi(b, c)
  \]

  \[
  e = \phi(b, d)
  \]

  (a) **SSA form**

  \[
  @a = \phi(@b, @c)
  \]

  \[
  @e = \phi(@b, @d)
  \]

  (b) **Spill**

  \[
  v_1 = ld@b
  \]

  \[
  @ac = st \ v_1
  \]

  \[
  v_2 = ld@b
  \]

  \[
  @ed = st \ v_2
  \]

  (c) **Coalescing and repairing**
Partial Rematerialization Support

(a) Origin

```
while(...){
    a = remat
    = a
    ⌞
}
```

(b) Partial support

```
while(...){
    a = remat
    = a
    ⌞
    a = remat
}
```

(c) Optimal

```
while(...){
    a = remat
    = a
    ⌞
}
```
Partial SSA Support: Ebner et al.

- No particular constraints on $\phi$-operations.
- Deal with $\phi$-operations with mixed type of operands.
- $\Rightarrow$ Repairing cost not in the model.

Example:

\[
\begin{align*}
\mathtt{a} & = \phi(\mathtt{b}, \mathtt{c}) \\
\mathtt{e} & = \phi(\mathtt{b}, \mathtt{d})
\end{align*}
\]
Program Point and ILP Variables

store a
\( l_{p,a} = ? \quad s_{p,a} = ? \)

\( \rho_{p,a} =? \quad \mu_{p,a} =? \)

load a

\( \bar{\rho}_{p,a} = 1 \quad \bar{\mu}_{p,a} =? \)

\( b = a + 1 \)

(a) A program point and its ILP variables

\( \bullet p \)

\( b = a + 1 \geq \geq \)

\( \rho_{q,b} = 1 \quad \mu_{q,b} = 0 \quad \bullet q \)

\( \rho_{q,a} =? \quad \mu_{q,a} =? \)

(b) Program points surrounding an instruction
Emulating other Approaches

Constraints to emulate Appel & George:

\[(\text{Appel}) \ \bar{\mu}_{p,v} + \bar{\rho}_{p,v} = 1\]

Alternatively:

\[(\text{Appel}_l) \ l_{p,v} + \bar{\mu}_{p,v} \leq 1 \quad (\text{Appel}_s) \ s_{p,v} + \bar{\rho}_{p,v} \leq 1\]

Constraints to emulate Koes & Goldstein:

\[(\text{Appel}_s) \ s_{p,v} + \bar{\rho}_{p,v} \leq 1\]
Discussion

- Huge gains in spill costs
  - Compared to 'optimal' techniques
  - Mostly due to elimination of stores

- Dynamic metrics
  - Lower cache miss rates
  - Lower number of loads/stores (−20%)
  - Lower number of operations executed (−8%)
  - Lower number of instruction bundles

- Marginal improvements in actual runtime
  - Costs of stores 'over-weighted'
  - Costs of secondary effects are missing
    (pipeline, cache, code layout)
Optimal Coalescing

\[
\begin{align*}
a = \ldots & \quad \text{store } a \text{ at } @c \\
b = \ldots & \quad \text{store } b \text{ at } @b \\
\text{if } (...) & \quad \text{store } b \text{ at } @c \\
c = \phi(a, b) & \quad \text{mem\_dup } c = b \\
\end{align*}
\]

(a) Original

\[
\begin{align*}
& \quad \text{load } b \\
& \quad \text{if } (...) \\
& \quad \text{store } b \text{ at } @c \\
c = \phi(a, b) & \\
\end{align*}
\]

(b) Optimistic/pessimistic

\[
\begin{align*}
& \quad \text{mem\_dup } c = b \\
& \quad \text{endif} \\
& \quad c = \phi(a, b) \\
\end{align*}
\]

(c) Optimal