Cache- and IO-Efficient Functional Algorithms

Guy E. Blelloch Robert Harper

Computer Science Department
Carnegie Mellon University

HCSS 2013
(Thanks to Byron Cook for the invitation.)

Languages and Algorithms

Algorithm analysis is based on low-level machine models.

- Time = number of instructions.
- Space = number of cells of storage.

Languages and Algorithms

Algorithm analysis is based on low-level machine models.

- Time = number of instructions.
- Space = number of cells of storage.

Machine-based approaches suffer some important weaknesses:

- Relies on pseudo-code and compilation strategy.
- Not very realistic, eg with respect to memory hierarchies.
- No concept of composition of programs.

Languages and Algorithms

Our goal is to promote functional language models for algorithms.

- Replace pseudo-code by real code.
- Analyze the code you actually run.
- Independent of a compilation method.

Languages and Algorithms

Our goal is to promote functional language models for algorithms.

- Replace pseudo-code by real code.
- Analyze the code you actually run.
- Independent of a compilation method.

Cost Semantics: associate an abstract cost with each execution.

Languages and Algorithms

Our goal is to promote functional language models for algorithms.

- Replace pseudo-code by real code.
- Analyze the code you actually run.
- Independent of a compilation method.

Cost Semantics: associate an abstract cost with each execution.

Provable Implementation: map abstract cost to concrete cost on a machine with provable performance bound.

Languages and Algorithms

Our goal is to promote functional language models for algorithms.

- Replace pseudo-code by real code.
- Analyze the code you actually run.
- Independent of a compilation method.

Cost Semantics: associate an abstract cost with each execution.

Provable Implementation: map abstract cost to concrete cost on a machine with provable performance bound.

Obtain end-to-end asymptotics for realistic functional languages.

Example: Parallelism [B & Greiner 96]

Associate a dynamic dependency graph to an evaluation derivation.

- Records true data dependencies (no approximation).
- Exposes inherent parallelism and sequentiality.

Example: Parallelism [B & Greiner 96]

Associate a dynamic dependency graph to an evaluation derivation.

- Records true data dependencies (no approximation).
- Exposes inherent parallelism and sequentiality.

Two measures of a cost graph g:

- Work, or sequential complexity: size of g.
- Span, or parallel complexity: diameter of g.

Example: Parallelism [B & Greiner 96]

Example: function application.

$$\frac{e_1 \Downarrow \quad \lambda x.e \quad e_2 \Downarrow \quad v_2 \quad [v_2/x]e \Downarrow \quad v}{e_1(e_2) \Downarrow \qquad\qquad v}$$

Example: Parallelism [B & Greiner 96]

Example: function application.

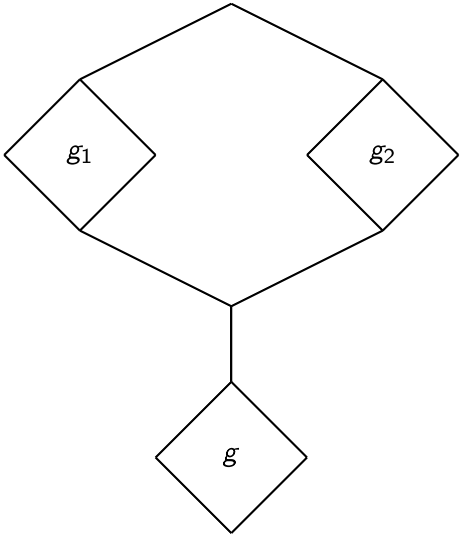
$$\frac{e_1 \Downarrow^{g_1} \lambda x.e \quad e_2 \Downarrow^{g_2} v_2 \quad [v_2/x]e \Downarrow^{g} v}{e_1(e_2) \Downarrow \qquad\qquad v}$$

Example: Parallelism [B & Greiner 96]

Example: function application.

$$\frac{e_1 \Downarrow^{g_1} \lambda x.e \quad e_2 \Downarrow^{g_2} v_2 \quad [v_2/x]e \Downarrow^{g} v}{e_1(e_2) \Downarrow^{(g_1 \otimes g_2) \oplus \mathbf{1} \oplus g} v}$$

Cost Graphs



Work $= w_1 + w_2 + w + 1$, Span $= \max(s_1, s_2) + 1 + s$.

Provable Implementation

Brent's Theorem: A computation with work w and span s can be implemented on a p-processor PRAM in time $O(w/p + s)$.

- Work in chunks of p as much as possible.
- Proof is constructive: exhibits a scheduler.

Provable Implementation

Brent's Theorem: A computation with work w and span s can be implemented on a p-processor PRAM in time $O(w/p + s)$.

- Work in chunks of p as much as possible.
- Proof is constructive: exhibits a scheduler.

Validates prediction given by high-level asymptotics.

- Transfers from high-level to low-level model.
- Provable cost bounds on a PRAM.

IO Model [Aggarwal & Vitter 88]

RAM-based IO model:

- Unbounded secondary memory, bounded primary memory.
- Cost = blocked transfers between primary and secondary.

IO Model [Aggarwal & Vitter 88]

RAM-based IO model:

- Unbounded secondary memory, bounded primary memory.
- Cost = blocked transfers between primary and secondary.

Example results:

- Matrix multiply without blocking: $O(n^3/B)$.
- …with blocking: $O(n^3/(B\sqrt{M})$.
- 2-way merge sort: $O((n/B)\log_2(n/B))$.
- …M/B-way: $O((n/B)\log_{(M/B)}(n/B))$

IO Model [Aggarwal & Vitter 88]

RAM-based IO model:

- Unbounded secondary memory, bounded primary memory.
- Cost = blocked transfers between primary and secondary.

Example results:

- Matrix multiply without blocking: $O(n^3/B)$.
- …with blocking: $O(n^3/(B\sqrt{M})$.
- 2-way merge sort: $O((n/B)\log_2(n/B))$.
- … M/B-way: $O((n/B)\log_{(M/B)}(n/B))$

Memory allocation and layout done by hand!

IO Efficient Functional Algorithms

Replicate A&V results in a purely functional language model.

- Automatic storage management.
- Natural functional code, not pseudo-code.

IO Efficient Functional Algorithms

Replicate A&V results in a purely functional language model.

- Automatic storage management.
- Natural functional code, not pseudo-code.

Key ideas:

- Operations in primary memory are cost-free.

IO Efficient Functional Algorithms

Replicate A&V results in a purely functional language model.

- Automatic storage management.
- Natural functional code, not pseudo-code.

Key ideas:

- Operations in primary memory are cost-free.
- Charge only for migration to and from secondary memory.

IO Efficient Functional Algorithms

Replicate A&V results in a purely functional language model.

- Automatic storage management.
- Natural functional code, not pseudo-code.

Key ideas:

- Operations in primary memory are cost-free.
- Charge only for migration to and from secondary memory.
- Provably efficient implementation on A & V machine model.

IO Efficient Functional Algorithms

Replicate A&V results in a purely functional language model.

- Automatic storage management.
- Natural functional code, not pseudo-code.

Key ideas:

- Operations in primary memory are cost-free.
- Charge only for migration to and from secondary memory.
- Provably efficient implementation on A & V machine model.

IO Efficient Functional Algorithms

Replicate A&V results in a purely functional language model.

- Automatic storage management.
- Natural functional code, not pseudo-code.

Key ideas:

- Operations in primary memory are cost-free.
- Charge only for migration to and from secondary memory.
- Provably efficient implementation on A & V machine model.

Confirms that automatic storage management is cache-friendly.

Simplified Cost Semantics for IO

Evaluation: $\sigma \,@\, e \Downarrow^{n} \sigma' \,@\, l$.

- All values are allocated at a location in storage.

Simplified Cost Semantics for IO

Evaluation: $\sigma \mathbin{@} e \Downarrow^{n} \sigma' \mathbin{@} l$.

- All values are allocated at a location in storage.
- Cost n measures traffic between primary and secondary.

Simplified Cost Semantics for IO

Evaluation: $\sigma \mathbin{@} e \Downarrow^{n} \sigma' \mathbin{@} l$.

- All values are allocated at a location in storage.
- Cost n measures traffic between primary and secondary.

Simplified Cost Semantics for IO

Evaluation: $\sigma \,@\, e \Downarrow^n \sigma' \,@\, l$.

- All values are allocated at a location in storage.
- Cost n measures traffic between primary and secondary.

Storage model: $\sigma = (\mu, \rho, \nu)$ [Morrisett, Felleisen, & H. 95]

- μ: unbounded secondary memory with blocks of size B.

Simplified Cost Semantics for IO

Evaluation: $\sigma \mathbin{@} e \Downarrow^{n} \sigma' \mathbin{@} l$.

- All values are allocated at a location in storage.
- Cost n measures traffic between primary and secondary.

Storage model: $\sigma = (\mu, \rho, \nu)$ [Morrisett, Felleisen, & H. 95]

- μ: unbounded secondary memory with blocks of size B.
- ρ: bounded primary memory of size $M = k \times B$.

Simplified Cost Semantics for IO

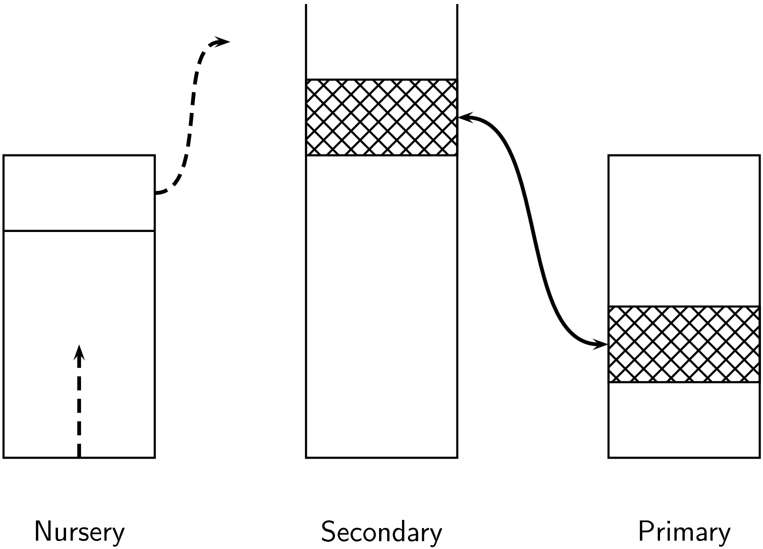
Evaluation: $\sigma @ e \Downarrow^{n} \sigma' @ l$.

- All values are allocated at a location in storage.
- Cost n measures traffic between primary and secondary.

Storage model: $\sigma = (\mu, \rho, \nu)$ [Morrisett, Felleisen, & H. 95]

- μ: unbounded secondary memory with blocks of size B.
- ρ: bounded primary memory of size $M = k \times B$.
- ν: nursery of size M with a linear ordering on its domain.

Simplified Memory Model



Simplified Cost Semantics

Read: $\sigma \,\texttt{@}\, l \downarrow^{n} \sigma' \,\texttt{@}\, v$.

- Read location l from store σ to obtain value v.
- Cost accounts for loads to and evictions from primary.
- Eviction policy by the Ideal Cache Model.

Simplified Cost Semantics

Read: $\sigma \mathbin{@} l \downarrow^{n} \sigma' \mathbin{@} v$.

- Read location l from store σ to obtain value v.
- Cost accounts for loads to and evictions from primary.
- Eviction policy by the Ideal Cache Model.

Allocate: $\sigma \mathbin{@} v \uparrow^{n} \sigma' \mathbin{@} l$.

- Allocate value v in σ obtaining σ' and l.
- Cost n accounts for migration to secondary.
- Live objects are blocked on migration to secondary.

Simplified Cost Semantics

Functions are allocated in memory:

$$\frac{\sigma \mathbin{@} \lambda x.e \uparrow^{n} \; \sigma' \mathbin{@} l}{\sigma \mathbin{@} \lambda x.e \Downarrow^{n} \; \sigma' \mathbin{@} l}$$

Simplified Cost Semantics

Functions are allocated in memory:

$$\frac{\sigma \mathbin{@} \lambda x.e \uparrow^{n} \sigma' \mathbin{@} l}{\sigma \mathbin{@} \lambda x.e \Downarrow^{n} \sigma' \mathbin{@} l}$$

Application follows pointers:

$$\frac{\left\{ \quad \sigma_1 \mathbin{@} e_1 \Downarrow^{n'_1} \quad \sigma'_1 \mathbin{@} l'_1 \right\}}{\sigma \mathbin{@} \mathtt{app}(e_1; e_2) \Downarrow^{n'_1+n''_1+ \quad n_2+n'_2} \sigma' \mathbin{@} l'}$$

Simplified Cost Semantics

Functions are allocated in memory:

$$\frac{\sigma \mathbin{@} \lambda x.e \uparrow^{n} \ \sigma' \mathbin{@} l}{\sigma \mathbin{@} \lambda x.e \Downarrow^{n} \ \sigma' \mathbin{@} l}$$

Application follows pointers:

$$\frac{\left\{ \begin{array}{ll} & \sigma_1 \mathbin{@} e_1 \Downarrow^{n_1'} \qquad \sigma_1' \mathbin{@} l_1' \\ \sigma_1' \mathbin{@} l_1' \downarrow^{n_1''} \sigma_1'' \mathbin{@} \lambda x.e & \\ & \end{array} \right\}}{\sigma \mathbin{@} \mathtt{app}(e_1; e_2) \Downarrow^{n_1' + n_1'' + \qquad n_2 + n_2'} \sigma' \mathbin{@} l'}$$

Simplified Cost Semantics

Functions are allocated in memory:

$$\frac{\sigma \mathbin{\mathtt{@}} \lambda x.e \uparrow^{n} \ \sigma' \mathbin{\mathtt{@}} l}{\sigma \mathbin{\mathtt{@}} \lambda x.e \Downarrow^{n} \ \sigma' \mathbin{\mathtt{@}} l}$$

Application follows pointers:

$$\frac{\left\{\begin{array}{lll} & & \sigma_1 \mathbin{\mathtt{@}} e_1 \Downarrow^{n_1'} \qquad \sigma_1' \mathbin{\mathtt{@}} l_1' \\ \sigma_1' \mathbin{\mathtt{@}} l_1' \downarrow^{n_1''} \sigma_1'' \mathbin{\mathtt{@}} \lambda x.e & & \\ \sigma_1'' \mathbin{\mathtt{@}} e_2 \Downarrow^{n_2} \qquad \sigma_2' \mathbin{\mathtt{@}} l_2' & & \end{array}\right\}}{\sigma \mathbin{\mathtt{@}} \mathtt{app}(e_1; e_2) \Downarrow^{\quad n_1'+n_1''+ \qquad n_2+n_2'} \sigma' \mathbin{\mathtt{@}} l'}$$

Simplified Cost Semantics

Functions are allocated in memory:

$$\frac{\sigma \mathbin{@} \lambda x.e \uparrow^{n} \sigma' \mathbin{@} l}{\sigma \mathbin{@} \lambda x.e \Downarrow^{n} \sigma' \mathbin{@} l}$$

Application follows pointers:

$$\frac{\left\{\begin{array}{lll} & \sigma_1 \mathbin{@} e_1 \Downarrow^{n_1'} & \sigma_1' \mathbin{@} l_1' \\ \sigma_1' \mathbin{@} l_1' \downarrow^{n_1''} \sigma_1'' \mathbin{@} \lambda x.e & & \\ \sigma_1'' \mathbin{@} e_2 \Downarrow^{n_2} \quad \sigma_2' \mathbin{@} l_2' & \sigma_2' \mathbin{@} [l_2'/x]e \Downarrow^{n_2'} & \sigma' \mathbin{@} l' \end{array}\right\}}{\sigma \mathbin{@} \mathtt{app}(e_1; e_2) \Downarrow^{n_1'+n_1''+n_2+n_2'} \sigma' \mathbin{@} l'}$$

Example: Map

Mapping over a list:

```
fun map f nil = nil
  | map f (h::t) = (f t) :: map f t
```

Example: Map

Mapping over a list:

```
fun map f nil = nil
  | map f (h::t) = (f t) :: map f t
```

Definition A list is compact if it can be traversed in time $O(n/B)$.

- Intuitively, not scattered through memory.
- Robust with respect to forward or backward traversal.

Example: Map

Mapping over a list:

```
fun map f nil = nil
  | map f (h::t) = (f t) :: map f t
```

Definition A list is compact if it can be traversed in time $O(n/B)$.

- Intuitively, not scattered through memory.
- Robust with respect to forward or backward traversal.

Theorem If l is compact and f is simple, then `map` f l is compact and has IO cost $O(n/B)$.

Example: Merge

Nearly standard implementation:

```
fun merge nil ys = ys
  | merge xs nil = xs
  | merge (xs as x::xs’) (ys as y::ys’) =
    case compare x y of
      LESS ⇒ !x::merge xs’ ys
    | GTEQ ⇒ !y::merge xs ys’
```

Example: Merge

Nearly standard implementation:

```
fun merge nil ys = ys
  | merge xs nil = xs
  | merge (xs as x::xs') (ys as y::ys') =
    case compare x y of
      LESS ⇒ !x::merge xs' ys
    | GTEQ ⇒ !y::merge xs ys'
```

The notations `!x` and `!x` denote deep copy to ensure compactness.

Example: Merge Sort

Theorem For compact inputs of size n and simple comparison, `merge xs ys` has cost $O(n/B)$.

Example: Merge Sort

Theorem For compact inputs of size n and simple comparison, `merge xs ys` has cost $O(n/B)$.

- Recurs down lists allocating only stack n frames: $O(n/B)$.
- Returns allocating n list cells: $O(n/B)$.

Example: Merge Sort

Theorem For compact inputs of size n and simple comparison, `merge xs ys` has cost $O(n/B)$.

- Recurs down lists allocating only stack n frames: $O(n/B)$.
- Returns allocating n list cells: $O(n/B)$.

Theorem For compact input of size n, `sort xs` has cost $O((n/B) \log_{(M/B)}(n/B))$.

Example: Merge Sort

Theorem For compact inputs of size n and simple comparison, `merge xs ys` has cost $O(n/B)$.

- Recurs down lists allocating only stack n frames: $O(n/B)$.
- Returns allocating n list cells: $O(n/B)$.

Theorem For compact input of size n, `sort xs` has cost $O((n/B) \log_{(M/B)}(n/B))$.

(Matches A&V bound in IO model.)

Provable Implementation (First Attempt)

"Theorem" If $\sigma \mathbin{@} e \Downarrow^{n} \sigma' \mathbin{@} l$, then e may be evaluated in the IO model in time $k \times n$ using a primary memory of size $4 \times M$.

Provable Implementation (First Attempt)

"Theorem" If $\sigma \mathbin{@} e \Downarrow^{n} \sigma' \mathbin{@} l$, then e may be evaluated in the IO model in time $k \times n$ using a primary memory of size $4 \times M$.

Proof sketch:

- Copying GC with semispaces for nursery: $2 \times M$.

Provable Implementation (First Attempt)

"Theorem" If $\sigma \mathbin{@} e \Downarrow^n \sigma' \mathbin{@} l$, then e may be evaluated in the IO model in time $k \times n$ using a primary memory of size $4 \times M$.

Proof sketch:

- Copying GC with semispaces for nursery: $2 \times M$.
- LRU is 2-competitive with ICM [Sleator & Tarjan 85]: $2 \times M$.

Provable Implementation (First Attempt)

“Theorem” If $\sigma \mathbin{@} e \Downarrow^{n} \sigma' \mathbin{@} l$, then e may be evaluated in the IO model in time $k \times n$ using a primary memory of size $4 \times M$.

Proof sketch:

- Copying GC with semispaces for nursery: $2 \times M$.
- LRU is 2-competitive with ICM [Sleator & Tarjan 85]: $2 \times M$.

Provable Implementation (First Attempt)

"Theorem" If $\sigma \mathbin{@} e \Downarrow^{n} \sigma' \mathbin{@} l$, then e may be evaluated in the IO model in time $k \times n$ using a primary memory of size $4 \times M$.

Proof sketch:

- Copying GC with semispaces for nursery: $2 \times M$.
- LRU is 2-competitive with ICM [Sleator & Tarjan 85]: $2 \times M$.

However, the "theorem" is not quite correct as stated . . . .

Stack Management

Simplified semantics does not account for control stack.

- For `map` stack space can be amortized against allocation (DPS).
- But a program may use more stack space than data space!

Stack Management

Simplified semantics does not account for control stack.

- For `map` stack space can be amortized against allocation (DPS).
- But a program may use more stack space than data space!

But consider non-tail recursive factorial:

```
fun fact 0 = 1
  | fact n = n * fact (n-1)
```

Stack Management

Simplified semantics does not account for control stack.

- For `map` stack space can be amortized against allocation (DPS).
- But a program may use more stack space than data space!

But consider non-tail recursive factorial:

```
fun fact 0 = 1
  | fact n = n * fact (n-1)
```

Simplified semantics predicts $O(1)$ cost, but true cost is $O(n/B)$!

Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

$$\frac{\left\{ \begin{array}{l} \sigma \mathbin{@} \mathtt{app}(-; e_2) \uparrow^{n_1}_{R \cup \mathrm{locs}(e_1)} \sigma_1 \mathbin{@} k_1 \\ \\ \\ \end{array} \right\}}{\sigma \mathbin{@} \mathtt{app}(e_1; e_2) \Downarrow^{n_1 + n_1' + n_1'' + n_1''' + n_2 + n_2'}_{R} \sigma' \mathbin{@} l'}$$

Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

$$\frac{\left\{ \begin{array}{l} \sigma \mathbin{@} \mathtt{app}(-; e_2) \uparrow^{n_1}_{R \cup \mathrm{locs}(e_1)} \sigma_1 \mathbin{@} k_1 \quad \sigma_1 \mathbin{@} e_1 \Downarrow^{n_1'}_{R \cup \{k_1\}} \sigma_1' \mathbin{@} l_1' \\ \\ \\ \end{array} \right\}}{\sigma \mathbin{@} \mathtt{app}(e_1; e_2) \Downarrow^{n_1 + n_1' + n_1'' + n_1''' + n_2 + n_2'}_{R} \sigma' \mathbin{@} l'}$$

Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

$$\frac{\left\{\begin{array}{l} \sigma \mathbin{\texttt{@}} \texttt{app}(-;e_2) \uparrow^{n_1}_{R \cup \mathrm{locs}(e_1)} \sigma_1 \mathbin{\texttt{@}} k_1 \quad \sigma_1 \mathbin{\texttt{@}} e_1 \Downarrow^{n_1'}_{R \cup \{k_1\}} \sigma_1' \mathbin{\texttt{@}} l_1' \\ \quad \sigma_1' \mathbin{\texttt{@}} l_1' \downarrow^{n_1''} \sigma_1'' \mathbin{\texttt{@}} \lambda x.e \\ \\ \end{array}\right\}}{\sigma \mathbin{\texttt{@}} \texttt{app}(e_1;e_2) \Downarrow_R^{n_1+n_1'+n_1''+n_1'''+n_2+n_2'} \sigma' \mathbin{\texttt{@}} l'}$$

Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

$$\frac{\left\{\begin{array}{cc} \sigma \mathbin{@} \mathtt{app}(-; e_2) \uparrow^{n_1}_{R\cup\mathrm{locs}(e_1)} \sigma_1 \mathbin{@} k_1 & \sigma_1 \mathbin{@} e_1 \Downarrow^{n_1'}_{R\cup\{k_1\}} \sigma_1' \mathbin{@} l_1' \\ \sigma_1' \mathbin{@} l_1' \downarrow^{n_1''} \sigma_1'' \mathbin{@} \lambda x.e & \sigma_1'' \mathbin{@} \mathtt{app}(l_1'; -) \uparrow^{n_1'''}_{R} \sigma_2 \mathbin{@} k_2 \\ & \end{array}\right\}}{\sigma \mathbin{@} \mathtt{app}(e_1; e_2) \Downarrow^{n_1+n_1'+n_1''+n_1'''+n_2+n_2'}_{R} \sigma' \mathbin{@} l'}$$

Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

$$\frac{\left\{\begin{array}{cc} \sigma \mathbin{@} \mathtt{app}(-;e_2) \uparrow^{n_1}_{R\cup \mathrm{locs}(e_1)} \sigma_1 \mathbin{@} k_1 & \sigma_1 \mathbin{@} e_1 \Downarrow^{n_1'}_{R\cup\{k_1\}} \sigma_1' \mathbin{@} l_1' \\ \sigma_1' \mathbin{@} l_1' \downarrow^{n_1''} \sigma_1'' \mathbin{@} \lambda x.e & \sigma_1'' \mathbin{@} \mathtt{app}(l_1';-) \uparrow^{n_1'''}_{R} \sigma_2 \mathbin{@} k_2 \\ \sigma_2 \mathbin{@} e_2 \Downarrow^{n_2}_{R\cup\{k_2\}} \sigma_2' \mathbin{@} l_2' & \end{array}\right\}}{\sigma \mathbin{@} \mathtt{app}(e_1;e_2) \Downarrow^{n_1+n_1'+n_1''+n_1'''+n_2+n_2'}_{R} \sigma' \mathbin{@} l'}$$

Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

$$\frac{\left\{\begin{array}{cc} \sigma \mathbin{@} \mathrm{app}(-; e_2) \uparrow^{n_1}_{R \cup \mathrm{locs}(e_1)} \sigma_1 \mathbin{@} k_1 & \sigma_1 \mathbin{@} e_1 \Downarrow^{n_1'}_{R \cup \{k_1\}} \sigma_1' \mathbin{@} l_1' \\ \sigma_1' \mathbin{@} l_1' \downarrow^{n_1''} \sigma_1'' \mathbin{@} \lambda x.e & \sigma_1'' \mathbin{@} \mathrm{app}(l_1'; -) \uparrow^{n_1'''}_{R} \sigma_2 \mathbin{@} k_2 \\ \sigma_2 \mathbin{@} e_2 \Downarrow^{n_2}_{R \cup \{k_2\}} \sigma_2' \mathbin{@} l_2' & \sigma_2' \mathbin{@} [l_2'/x]e \Downarrow^{n_2'}_{R} \sigma' \mathbin{@} l' \end{array}\right\}}{\sigma \mathbin{@} \mathrm{app}(e_1; e_2) \Downarrow^{n_1 + n_1' + n_1'' + n_1''' + n_2 + n_2'}_{R} \sigma' \mathbin{@} l'}$$

Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

$$\frac{\left\{\begin{array}{cc} \sigma \mathbin{@} \mathrm{app}(-;e_2) \uparrow^{n_1}_{R\cup\mathrm{locs}(e_1)} \sigma_1 \mathbin{@} k_1 & \sigma_1 \mathbin{@} e_1 \Downarrow^{n_1'}_{R\cup\{k_1\}} \sigma_1' \mathbin{@} l_1' \\ \sigma_1' \mathbin{@} l_1' \downarrow^{n_1''} \sigma_1'' \mathbin{@} \lambda x.e & \sigma_1'' \mathbin{@} \mathrm{app}(l_1';-) \uparrow^{n_1'''}_{R} \sigma_2 \mathbin{@} k_2 \\ \sigma_2 \mathbin{@} e_2 \Downarrow^{n_2}_{R\cup\{k_2\}} \sigma_2' \mathbin{@} l_2' & \sigma_2' \mathbin{@} [l_2'/x]e \Downarrow^{n_2'}_{R} \sigma' \mathbin{@} l' \end{array}\right\}}{\sigma \mathbin{@} \mathrm{app}(e_1;e_2) \Downarrow_R^{n_1+n_1'+n_1''+n_1'''+n_2+n_2'} \sigma' \mathbin{@} l'}$$

Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

$$\frac{\left\{\begin{array}{cc} \sigma \mathbin{\mathtt{@}} \mathrm{app}(-;e_2) \uparrow^{n_1}_{R\cup\mathrm{locs}(e_1)} \sigma_1 \mathbin{\mathtt{@}} k_1 & \sigma_1 \mathbin{\mathtt{@}} e_1 \Downarrow^{n_1'}_{R\cup\{k_1\}} \sigma_1' \mathbin{\mathtt{@}} l_1' \\ \sigma_1' \mathbin{\mathtt{@}} l_1' \downarrow^{n_1''} \sigma_1'' \mathbin{\mathtt{@}} \lambda x.e & \sigma_1'' \mathbin{\mathtt{@}} \mathrm{app}(l_1';-) \uparrow^{n_1'''}_{R} \sigma_2 \mathbin{\mathtt{@}} k_2 \\ \sigma_2 \mathbin{\mathtt{@}} e_2 \Downarrow^{n_2}_{R\cup\{k_2\}} \sigma_2' \mathbin{\mathtt{@}} l_2' & \sigma_2' \mathbin{\mathtt{@}} [l_2'/x]e \Downarrow^{n_2'}_{R} \sigma' \mathbin{\mathtt{@}} l' \end{array}\right\}}{\sigma \mathbin{\mathtt{@}} \mathrm{app}(e_1;e_2) \Downarrow^{n_1+n_1'+n_1''+n_1'''+n_2+n_2'}_{R} \sigma' \mathbin{\mathtt{@}} l'}$$

Modifications:

- Frames are never read! Allocation cost suffices.
- Root set R records live data in the (implicit) control stack.

Provable Implementation (Corrected)

Theorem If $\sigma \mathbin{@} e \Downarrow^{n}_{R} \sigma' \mathbin{@} l$, then e can be executed in the IO model in time $k \times n$ using a primary cache of size $4 \times M + B$.

Provable Implementation (Corrected)

Theorem If $\sigma \mathbin{@} e \Downarrow^{n}_{R} \sigma' \mathbin{@} l$, then e can be executed in the IO model in time $k \times n$ using a primary cache of size $4 \times M + B$.

Proof given in two major steps:

- Implement cost semantics on a stack machine.
- Implement stack machine on A&V IO model.

Stack Management

Stack frames are allocated in the nursery.

- May exist solely within nursery.
- May migrate to secondary memory.

Stack Management

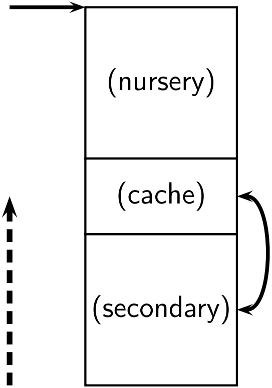
Stack frames are allocated in the nursery.

- May exist solely within nursery.
- May migrate to secondary memory.

Dedicate a cache block of B frames in primary memory.

- Not influenced by frames in nursery.
- Specially managed read cache for stack frames.

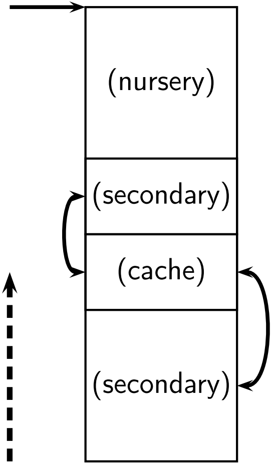
Stack Management


(nursery)

(cache)

(secondary)

Typical Stack


(nursery)

(secondary)

(cache)

(secondary)

Deep Recursion

Stack Management

Stack cache block may be evicted up to B times.

- Newer frames may overflow nursery.
- Reading evicted frames replaces stack cache.

Stack Management

Stack cache block may be evicted up to B times.

- Newer frames may overflow nursery.
- Reading evicted frames replaces stack cache.

Amortize cost of eviction over allocation of newer frames.

- Put \$3 on each frame block as it is migrated to secondary.
- Use \$1 for migration.
- Use \$1 for initial load.
- Use \$1 for reload of evicted block.

Summary

Cost semantics supports analysis of complexity of high-level code.

- No need for “pseudo-code”.

Summary

Cost semantics supports analysis of complexity of high-level code.

- No need for "pseudo-code".
- Avoid reasoning about implementation.

Summary

Cost semantics supports analysis of complexity of high-level code.

- No need for "pseudo-code".
- Avoid reasoning about implementation.

Summary

Cost semantics supports analysis of complexity of high-level code.

- No need for "pseudo-code".
- Avoid reasoning about implementation.

Aggarwal & Vitter's results can be matched using natural functional code.

- Must consider compactness of data structures.

Summary

Cost semantics supports analysis of complexity of high-level code.

- No need for “pseudo-code”.
- Avoid reasoning about implementation.

Aggarwal & Vitter’s results can be matched using natural functional code.

- Must consider compactness of data structures.
- End-to-end comparable to machine-level implementation.

Open Questions

Can we sort IO optimally with a cache oblivious algorithm?

- Merge sort uses M/B-way split.
- Frigo, et al. 99 give a cache-oblivious sorting algorithm.

Open Questions

Can we sort IO optimally with a cache oblivious algorithm?

- Merge sort uses M/B-way split.
- Frigo, et al. 99 give a cache-oblivious sorting algorithm.

Can the IO model be extended to account for parallelism?