Cache- and IO-Efficient Functional Algorithms

Guy E. Blelloch    Robert Harper

Computer Science Department
Carnegie Mellon University

POPL 2013
Languages and Algorithms

Algorithm analysis is based on low-level machine models.

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

\[ e_1 \downarrow \lambda x. e \quad e_2 \downarrow v_2 \quad [v_2/x] e \downarrow v \]
\[ e_1(e_2) \downarrow v \]
Example: function application.

\[
\begin{align*}
e_1 \Downarrow^{g_1} \lambda x.e & \quad e_2 \Downarrow^{g_2} v_2 & \quad [v_2/x]e \Downarrow^g v \\
\hline
e_1(e_2) \Downarrow & v
\end{align*}
\]
Example: Parallelism [B & Greiner 96]

Example: function application.

\[
\frac{e_1 \downarrow^{g_1} \lambda x. e \quad e_2 \downarrow^{g_2} v_2 \quad [v_2/x]e \downarrow^g v}{e_1(e_2) \downarrow^{(g_1 \otimes g_2) \oplus 1 \oplus g} v}
\]
Work = w_1 + w_2 + w + 1, Span = \max(s_1, s_2) + 1 + s.
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.
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.

Example results:

- Matrix multiply without blocking: $O\left(\frac{n^3}{B}\right)$.
- ...with blocking: $O\left(\frac{n^3}{B\sqrt{M}}\right)$.
- 2-way merge sort: $O\left((\frac{n}{B}) \log_2 (\frac{n}{B})\right)$.
- ...$\frac{M}{B}$-way: $O\left((\frac{n}{B}) \log \left(\frac{M}{B}\right) (\frac{n}{B})\right)$.

Memory allocation and layout done by hand!
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.
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**.
Evaluation: $\sigma @ e \Downarrow^n \sigma' @ l$.

- All values are allocated at a location in storage.
Simplified Cost Semantics for IO

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

- 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.
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]
- \( \mu \): unbounded secondary memory with blocks of size \( B \).
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]

- $\mu$: unbounded secondary memory with blocks of size $B$.
- $\rho$: 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]
- \( \mu \): unbounded secondary memory with blocks of size \( B \).
- \( \rho \): bounded primary memory of size \( M = k \times B \).
- \( \nu \): nursery of size \( M \) with a linear ordering on its domain.
Simplified Memory Model

Nursery

Secondary

Primary
Simplified Cost Semantics

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

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

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

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

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

- Allocate value $v$ in $\sigma$ obtaining $\sigma'$ 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:

\[
\sigma @ \lambda x. e \uparrow^n \sigma' @ l \\
\sigma @ \lambda x. e \downarrow^n \sigma' @ l
\]
Simplified Cost Semantics

Functions are allocated in memory:

\[
\begin{align*}
\sigma @ \lambda x. e & \uparrow^n \sigma' @ l \\
\sigma @ \lambda x. e & \downarrow^n \sigma' @ l
\end{align*}
\]

Application follows pointers:

\[
\begin{align*}
& \sigma_1 @ e_1 \downarrow^{n_1'} \\
& \sigma_1' @ l_1'
\end{align*}
\]

\[
\sigma @ \text{app}(e_1; e_2) \downarrow^{n_1' + n_1'' + n_2 + n_2'} \sigma' @ l''
\]
Simplified Cost Semantics

Functions are allocated in memory:

\[
\begin{align*}
\sigma @ \lambda x. e \uparrow^n & \sigma' @ l \\
\sigma @ \lambda x. e \downarrow^n & \sigma' @ l
\end{align*}
\]

Application follows pointers:

\[
\begin{align*}
\left\{ \begin{array}{c}
\sigma_1 @ e_1 \downarrow^{n_1'} \\
\sigma_1 @ l_1'
\end{array} \right\} \\
\sigma_1 @ l_1' \downarrow^{n_1''} \sigma_2 @ \lambda x. e
\end{align*}
\]

\[
\sigma @ \text{app}(e_1; e_2) \downarrow^{n_1' + n_1'' + n_2' + n_2''} \sigma' @ l''
\]
Simplified Cost Semantics

Functions are allocated in memory:

\[
\frac{\sigma \circ \lambda x.e \uparrow^n \sigma' \circ l}{\sigma \circ \lambda x.e \downarrow^n \sigma' \circ l}
\]

Application follows pointers:

\[
\begin{cases}
\sigma_1 \circ l_1 \downarrow^{n_1'} \sigma_1' \circ \lambda x.e \\
\sigma_2' \circ l_2 \\
\sigma_1 \circ \text{app}(e_1; e_2) \downarrow^{n_1'+n_2'} \sigma' \circ l'
\end{cases}
\]
Functions are allocated in memory:

\[
\begin{align*}
\sigma \circ \lambda x. e & \uparrow^n \sigma' \circ l \\
\sigma \circ \lambda x. e & \downarrow^n \sigma' \circ l
\end{align*}
\]

Application follows pointers:

\[
\begin{align*}
\sigma_1 \circ e_1 & \downarrow^{n_1'} \sigma_1' \circ l_1' \\
\sigma_1' \circ l_1' & \downarrow^{n_1''} \sigma_1'' \circ \lambda x. e \\
\sigma_1'' \circ e_2 & \downarrow^{n_2} \sigma_2' \circ l_2' \\
\sigma_2' \circ [l_2'/x] e & \downarrow^{n_2'} \sigma' \circ l'
\end{align*}
\]

\[
\sigma \circ \text{app}(e_1; e_2) \downarrow^{n_1' + n_1'' + n_2 + n_2'} \sigma' \circ l'
\]
Example: Map

Mapping over a list:

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

Mapping over a list:

```plaintext
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:

\[
\text{fun map } f \text{ nil } = \text{ nil} \\
| \text{map } f \text{ (h::t)} = (f \text{ t}) :: \text{map } f \text{ t}
\]

**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 $\text{map } f \ l$ is compact and has IO cost $O(n/B)$. 
Example: Merge

Nearly standard implementation:

```haskell
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

Nearly standard implementation:

```haskell
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 $\sigma \uparrow e \downarrow^n \sigma' \downarrow l$, 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 $\sigma \preceq e \downarrow^n \sigma' @ I$, 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: $2 \times M$. 
“Theorem” If $\sigma \circ e \Downarrow^n \sigma' \circ l$, 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]: $2 \times M$. 
“Theorem” If $\sigma @ e \downarrow^n \sigma' @ l$, 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]: $2 \times M$. 
“Theorem”  If \( \sigma \circ e \Downarrow^n \sigma' \circ l \), 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!

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

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

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

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

Simplified semantics predicts $O(1)$ cost, but true cost is $O(n/B)$!
The cost semantics must be enhanced to allocate frames:
The cost semantics must be enhanced to allocate frames:

\[
\sigma \odot \text{app}(e_1; e_2) \uparrow_{R \cup \text{locs}(e_1)}^{n_1} \sigma_1 \odot k_1
\]

\[
\sigma \odot \text{app}(e_1; e_2) \downarrow_R^{n_1 + n_1' + n_1'' + n_2 + n_2'} \sigma' \odot l'
\]
The cost semantics must be enhanced to allocate frames:

\[
\begin{align*}
\sigma \circ \text{app}(\_; e_2) \uparrow_{R \cup \text{locs}(e_1)}^{n_1} \sigma_1 \circ k_1 & & \sigma_1 \circ e_1 \downarrow_{R \cup \{ k_1 \}}^{n_1'} \sigma_1' \circ l'_1 \\
\sigma \circ \text{app}(e_1; e_2) \downarrow_{R}^{n_1 + n_1' + n_1'' + n_2 + n_2'} \sigma' \circ l''
\end{align*}
\]
Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

\[
\left\{ \begin{array}{ll}
\sigma \circ \text{app}(-; e_2) & \uparrow_{R \cup \text{locs}(e_1)}^{n_1} \sigma_1 \circ k_1 \\
\sigma_1 \circ e_1 & \downarrow_{R \cup \{k_1\}}^{n'_1} \sigma'_1 \circ l'_1 \\
\sigma'_1 \circ l'_1 & \downarrow_{n'_1}^{n''_1} \sigma''_1 \circ \lambda x. e \\
\sigma \circ \text{app}(e_1; e_2) & \downarrow_{R}^{n_1+n'_1+n''_1+n''''_1+n_2+n'_2} \sigma' \circ l''
\end{array} \right.
\]

Modifications:
• Frames are never read! Allocation cost suffices.
• Root set \( R \) records live data in the (implicit) control stack.
Cost Semantics for IO

The cost semantics must be enhanced to *allocate* frames:

\[
\begin{align*}
\sigma \circ \text{app}(e_1; e_2) \uparrow_{R \cup \text{locs}(e_1)}^{n_1} \quad & \sigma_1 \circ k_1 \\
\sigma_1 \circ e_1 \downarrow_{R \cup \{k_1\}}^{n_1'} \quad & \sigma_1' \circ l_1'' \\
\sigma_1' \circ l_1' \downarrow_{l_1'}^{n_1''} \quad & \sigma_1'' \circ \lambda x.e \\
\sigma_1'' \circ \text{app}(l_1'; -) \uparrow_{R}^{n_1'''} \quad & \sigma_2 \circ k_2
\end{align*}
\]

\[
\sigma \circ \text{app}(e_1; e_2) \downarrow_{R}^{n_1+n_1'+n_1''+n_1'''+n_2+n_2'} \quad \sigma' \circ l''
\]
The cost semantics must be enhanced to allocate frames:

\[
\begin{aligned}
\sigma @ \text{app}(\_; e_2) &\uparrow^{n_1}_{R \cup \text{locs}(e_1)} \sigma_1 @ k_1 & \sigma_1 @ e_1 &\downarrow^{n'_1}_{R \cup \{k_1\}} \sigma'_1 @ l'_1 \\
\sigma'_1 @ l'_1 &\downarrow^{n''_1} \sigma''_1 @ \lambda x. e & \sigma''_1 @ \text{app}(l'_1; \_) &\uparrow^{n'''_1}_{R} \sigma_2 @ k_2 \\
\sigma_2 @ e_2 &\downarrow^{n_2}_{R \cup \{k_2\}} \sigma'_2 @ l'_2 \\
\sigma @ \text{app}(e_1; e_2) &\downarrow^{n_1+n'_1+n''_1+n'''_1+n_2+n'_2}_{R} \sigma' @ l''
\end{aligned}
\]
Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

\[
\begin{align*}
\{ & \sigma \circ app(\_; e_2) \uparrow^{n_1}_{R \cup \text{locs}(e_1)} \sigma_1 \circ k_1 \quad \sigma_1 \circ e_1 \downarrow^{n'_1}_{R \cup \{k_1\}} \sigma'_1 \circ l'_1 \} \\
& \quad \{ \sigma'_1 \circ l'_1 \downarrow^{n''}_1 \sigma''_1 \circ \lambda x. e \quad \sigma''_1 \circ app(l'_1; \_) \uparrow^{n'''_1}_{R} \sigma_2 \circ k_2 \\
& \quad \{ \sigma_2 \circ e_2 \downarrow^{n_2}_{R \cup \{k_2\}} \sigma'_2 \circ l'_2 \\
& \quad \sigma'_2 \circ [l'_2/x]e \downarrow^{n'_2}_{R} \sigma' \circ l' \}
\}
\]

\[
\sigma \circ app(e_1; e_2) \downarrow^{n_1+n'_1+n''_1+n'''_1+n_2+n'_2}_{R} \sigma' \circ l'
\]
Cost Semantics for IO

The cost semantics must be enhanced to allocate frames:

\[
\begin{align*}
\{ 
\sigma \circ \text{app}(\_; e_2) & \uparrow_{R \cup \text{locs}(e_1)}^{n_1} \sigma_1 \circ k_1 & \sigma_1 \circ e_1 & \downarrow_{R \cup \{k_1\}}^{n_1'} \sigma_1' \circ l_1' \\
\sigma_1' \circ l_1' & \downarrow_{R \cup \{k_2\}}^{n_2} \sigma_1'' \circ \lambda x . e & \sigma_1'' \circ \text{app}(l_1'; \_ ) \uparrow_{R}^{n_1''} \sigma_2 \circ k_2 \\
\sigma_2 \circ e_2 & \downarrow_{R \cup \{k_2\}}^{n_2} \sigma_2' \circ l_2' & \sigma_2' \circ [l_2'/x]e & \downarrow_{R}^{n_2'} \sigma' \circ l'
\} \\
\sigma \circ \text{app}(e_1; e_2) & \downarrow_{R}^{n_1+n_1'+n_1''+n_1'''}+n_2+n_2' \sigma' \circ l'
\end{align*}
\]
The cost semantics must be enhanced to allocate frames:

$$
\begin{align*}
\sigma \circ \text{app}(\-; e_2) & \uparrow^{n_1}_{R \cup \text{locs}(e_1)} \sigma_1 \circ k_1 & \sigma_1 \circ e_1 & \downarrow^{n_1'}_{R \cup \{k_1\}} \sigma_1' \circ l_1' \\
\sigma_1' \circ l_1' & \downarrow^{n_{1''}} l_1' & \sigma_1'' \circ \lambda x. e & \sigma_1'' \circ \text{app}(l_1'; \-) \uparrow^{n_{1'''}}_{R} \sigma_2 \circ k_2 \\
\sigma_2 \circ e_2 & \downarrow^{n_2}_{R \cup \{k_2\}} \sigma_2' \circ l_2' & \sigma_2' \circ [l_2'/x] e & \downarrow^{n_2'}_{R} \sigma' \circ l' \\
\sigma \circ \text{app}(e_1; e_2) & \downarrow^{n_1+n_1'+n_{1''}+n_{1'''}+n_2+n_2'}_{R} \sigma' \circ l'
\end{align*}
$$

Modifications:

- Frames are never read! Allocation cost suffices.
- Root set $R$ records live data in the (implicit) control stack.
Theorem If $\sigma \circ e \Downarrow_R^n \sigma' \circ l$, then $e$ can be executed in the IO model in time $k \times n$ using a primary cache of size $4 \times M + B$. 
Theorem If $\sigma \circ e \Downarrow_R \sigma' \circ l$, 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 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

Typical Stack

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

- No need for “pseudo-code”.

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.
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.
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.
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?