# Cache Efficient Functional Algorithms

Robert Harper
Carnegie Mellon University
(With Guy E. Blelloch)

WG 2.8 Annapolis November 2012

# Machine Models

Traditionally, algorithm analysis is based on abstract machines.

- Classically, RAM or PRAM, with constant-time memory access.
- Low-level programming model, essentially assembly language.

Time complexity is measured by number of instruction steps.

- Robust across variations in model.
- Supports asymptotic time analysis.

# Machine Models

RAM model is unreasonably low-level.

- Manual memory management.
- No abstraction or composition.
- Write higher-level code, reason about its compilation.

Basic RAM model ignores memory hierarchy.

- Memory access time is not constant.
- Cache effects are significant.

# IO Model

Aggarwal and Vitter: I/O Model.

- Add cache of size $M = c \times B$ for some block size $B$.
- Memory traffic is in units of $B$ words.
- Analyze cache complexity.

Obtained matching lower and upper bounds for sorting.

- eg, $M/B$-way merge sort: $O((n/B) \log_{M/B}(n/B))$.
- (Not cache oblivious.)

# Language Models

We prefer to work with high-level linguistic models.

- Support abstraction and composition.
- Avoid low-level memory management and imperative mindset.

But can we understand their complexity?

- Avoid reasoning about compiled code.
- Account for implicit computation (esp., storage management).

# Functional Models

Computation by transformation, not mutation.

- Persistent data structures by default.
- Naturally parallel: no artificial dependencies, no contention.
- Easily verified by inductive arguments.

(The basis for introductory CS at CMU since 2010.)

# Functional Models

Functional mergesort:

```
fun mergesort xs =
  if size(xs) <= 1 then
    xs
  else
    let (xsl,xsr) = split xs in
      merge (mergesort xsl, mergesort xsr)
    end
```

As natural as one could imagine!

# Cost Semantics

Blelloch and Greiner pioneered a better way to go:

- Cost semantics: assign an abstract cost to a functional program.
- Provable implementation: transfer abstract costs to concrete costs.

Cost of execution is a series-parallel graph.

- Tracks dynamic data dependencies (no approximation).
- Work = size of graph = sequential complexity.
- Depth = span of graph = (idealized) parallel complexity.

# Implicit Parallelism

Evaluation: $e \Downarrow^{g} v$.

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

**Thm**: If $e \Downarrow^{g} v$, where $\mathrm{wk}(g) = w$ and $\mathrm{dp}(g) = d$, then $e$ may be evaluated on a $p$-processor PRAM in time $O(\max(w/p, d))$.

The proof encodes the scheduling strategy (Brent's Principle).

# Cache Complexity

Cost semantics for cache complexity is a bit more involved.

- Make reads and allocations explicit.
- In-cache computation costs 0; misses and evictions cost 1.
- Account for (implicit) control stack usage.

Provable implementation specifies:

- Stack management to control contention/interference (amortized analysis).
- Managing allocation and eviction (competitive analysis).

Main idea: ensure that temporal locality implies spatial locality.

# Cost Semantics Overview

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

- $\mu$ is an unbounded memory partitioned into blocks of size $B$.
- $\rho$ is a read cache of size $M$ partitioned into blocks of size $B$.
- $\nu$ is a nursery of size $M$ with a linear ordering $l_1 \prec_\nu l_2$.

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

- All values are allocated in the store, $\sigma$, at a location, $l$.
- Root set $R$ maintains liveness information.
- Abstract cost $n$ represents cache complexity.

# Cost Semantics Overview

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

- Read location $l$ from store $\sigma$ to obtain value $v$.
- Abstract cost $n$ represents cache loads and evictions.
- Store modification reflects cache effects.

Allocation: $\sigma \mathbin{@} v \uparrow^{n}_{R} \sigma' \mathbin{@} l$.

- Allocate value $v$ in $\sigma$ obtaining $\sigma'$ and new location $l$.
- Root set $R$ maintains liveness information.
- Abstract cost $n$ represents migration of objects to memory.

# Reading

In-cache and in-nursery reads are cost-free:

$$\frac{l \in \mathrm{dom}(\rho)}{(\mu, \rho, \nu) \mathbin{@} l \downarrow^{0} (\mu, \rho, \nu) \mathbin{@} \rho(l)}$$

$$\frac{l \in \mathrm{dom}(\nu)}{(\mu, \rho, \nu) \mathbin{@} l \downarrow^{0} (\mu, \rho, \nu) \mathbin{@} \nu(l)}$$

Out-of-cache reads load, and may evict, a block with cost $1/B$:

$$\frac{l \notin \mathrm{dom}(\rho) \cup \mathrm{dom}(\nu) \quad |\mathrm{dom}(\rho)| \leq M - B}{(\mu, \rho, \nu) \mathbin{@} l \downarrow^{1} (\mu, \rho \oplus \mathrm{nbhd}(\mu, l), \nu) \mathbin{@} \mu(l)}$$

$$\frac{l \notin \mathrm{dom}(\rho) \cup \mathrm{dom}(\nu) \quad |\mathrm{dom}(\rho)| = M \quad \beta \subseteq \rho}{(\mu, \rho, \nu) \mathbin{@} l \downarrow^{1} (\mu, \rho \ominus \beta \oplus \mathrm{nbhd}(\mu, l), \nu) \mathbin{@} \mu(l)}$$

# Allocation

Nursery limited to $M$ live objects:

$$\frac{|\operatorname{live}(R \cup \operatorname{locs}(o), \nu)| < M \quad l \notin \operatorname{dom}(\nu)}{(\mu, \rho, \nu) \mathbin{@} o \uparrow^{0}_{R} (\mu, \rho, \nu[l \mapsto o]) \mathbin{@} l}$$

Migration blocks $B$ oldest objects into memory:

$$\frac{|\operatorname{live}(R \cup \operatorname{locs}(o), \nu)| = M \quad \beta = \operatorname{scan}(R \cup \operatorname{locs}(o), \nu) \quad l \notin \operatorname{dom}(\nu)}{(\mu, \rho, \nu) \mathbin{@} o \uparrow^{1}_{R} (\mu \oplus \beta, \rho, (\nu \ominus \beta)[l \mapsto o]) \mathbin{@} l}$$

# Evaluation

Functions are allocated in storage, represented by a "pointer":

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

Application chases pointers and allocates frames:

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

# Critical Invariants

Stack frames are allocated to account for implicit storage:

- Maintains correct ordering of allocated space.
- Maintains liveness information within cache.

Object migration is oldest first:

- Migrate only live objects.
- Nursery is implicitly garbage-collected to free dead objects.
- Neighborhood is fixed at the moment of migration.

# Provable Implementation

Three main ingredients:

- Manage the memory traffic engendered by the control stack.
- Read cache eviction policy.
- Liveness analysis and compression for migration.

# Stack Management

Reserve a block of size $B$, the stack cache, for the top of the stack.

- Stack frames originate in the nursery, then migrate to memory as necessary.
- Stack frames are loaded into the stack cache as a block from main memory.
- Loading the stack cache evicts its current contents.

Must ensure that one block in the read cache is always available for the top of the control stack.

# Stack Management

Amortized analysis bounds cost of stack management:

- Accessing frames in the nursery is free.
- The first load of a frame must previously have been migrated to memory.
- Only newer frames can evict older frames from stack cache.
- Every frame must eventually be read and used exactly once.

Upshot: the traffic arising from stack frames may be attributed to their allocation.

# Stack Management

Associate the cost of the load and reload with the frames that force the eviction.

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

**Thm** A computation with abstract cache complexity $n$ can be implemented on a stack machine with cache complexity at most $3 \times n$.

# Allocation Management

Read and allocate may be implemented within a small constant, given a cache of size $4 \times M + B$ objects.

Storage assumptions:

- Object sizes are bounded by the size of the program.
- Must assume sufficient word size to hold a pointer.

Read cache evicts least-recently-used block.

- 2-competitive with ICM [Sleator, et al.]
- Standard, easily implemented.

# Allocation Management

Copying garbage collection manages liveness and compaction:

- Allocation of frames ensures that liveness can be determined without memory traffic.
- Require $2 \times M$ nursery size to allow for copying GC.
- Copying collection is constant-time per object (amortized across allocations).

Must double-load blocks to ensure that neighborhood is loaded even when GC is performed.

# Analysis Methods

A data structure of size $n$ is compact if it can be traversed in time $O(n/B)$ in the model.

- Intuitively, the components are allocated "adjacently."
- Robust under change of traversal order.
- Defined in the semantics, not the implementation.

A function is hereditarily finite (HF) if it maps hereditarily finite inputs to hereditarily finite outputs using only constant space.

- Used to analyze higher-order functions such as `map`.
- Standard notion in semantics.

# Example: Map

The `map` function transforms compact lists into compact lists.

- Temporal locality implies spatial locality.
- Assuming function mapped is hereditarily finite.

For HF $f$, `map f xs` has cache complexity $O(n/B)$, where $n$ is the length of `xs`.

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

# Example: Merge

Almost entirely standard implementation:

```
fun merge nil bs = bs
  | merge as nil = as
  | merge (as as a::as') (bs as b::bs') =
    case compare a b of
      LESS ⇒ !a::merge as' bs
    | GTEQ ⇒ !b::merge as bs'
```

Proviso: `!a` and `!b` specify copying of element to ensure compactness.

# Example: Merge

For HF `less` and compact `as` and `bs`, the function `merge as bs` has cache complexity $O(n/B)$, where $n$ is the sum of the sizes of `as` and `bs`.

Main points:

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

# Example: Merge Sort

A similar analysis for binary merge sort yields $O(n/B \log n/M)$ cache complexity.

- Same bound as for manual allocation.
- Cache oblivious: no use of cache parameters.

Aggarwal and Vitter's results on $k$-way merge are attained by the functional program.

- $O(n/B \log_{M/B} n/B)$.
- Not cache-oblivious: $k$ is $M/B$.

# Summary

Cost semantics mediates between analysis and implementation.

- High-level programming model: complexity analysis at the level of the code itself.
- Low-level implementation: transfer asymptotics to machine level.

The cache complexity of an algorithm may be expressed using a cost semantics for a purely functional language.

Can match bounds on cache complexity given for low-level models.