Concurrent Adaptive Computing
(Work in progress)

Matthew Hammer\textsuperscript{12}
hammer@tti-c.org

Joint work with
Umut Acar\textsuperscript{2}
Anwar Ghuloum\textsuperscript{1}
Leaf Petersen\textsuperscript{1}
Mohan Rajagopalan\textsuperscript{1}

\textsuperscript{1} Intel Programming Systems Lab
\textsuperscript{2} Toyota Technological Institute at Chicago

October 27, 2006
Big Picture (1)

\[ l_1 = \begin{array}{ccccc}
3 & 4 & 6 & 5 & 1 \\
\end{array} \]

\[ O_1 = \begin{array}{cccccc}
1 & 3 & 4 & 5 & 6 \\
\end{array} \]
**Big Picture (1)**

First, Change 6 to 2

Then, Propagate.

Now 6 is replaced by 2 in output and 2 appears in the correct position.
Adaptive programs

Definition
An adaptive program $P$ is a mapping from a set of inputs $I$ to a set of outputs $O$ where

- $P$ can be run “from scratch” on an input $I_1$ to get an initial output $P[I_1] = O_1$.
- Then given some new but related input $I_2$, change-propagation will compute $P[I_2] = O_2$ via reuse of the previous computation of $O_1$. 
What is $P$?, how do we make it?

- $P$ should be constructible given any suitable algorithm $A$
- $P$ shouldn’t run more than $O(1)$ worse than $A$
- When changes to input are small, $P$ should outperform $A$
Making program $P$ from program $A$

Perhaps two approaches:

- **A compiler plus a runtime**
  Future work for someone.

- **An API with a runtime.**
  My work, Acar et. al.’s work to date.
  The user writes their program against an API to yield the adaptive analogue $P$ of their original algorithm $A$. 
Outline

1. Introduction
   - Contributions
   - Current approach, limitations
   - Our solution

2. Program semantics

3. Change-propagation semantics

4. Summary, future work
Contributions

Starting point: Serial framework (Acar et al)

- is strictly serial
- relies on higher-order functions (with currying, closures, *etc* to use adaptive primitives.
- is formalized in limiting ways (destinations for *writes* are implicit, no support for multiple destinations).

Our extensions

- permit restricted forms of concurrency.
- are realized in C.
- have an associated formalism (still in works) which uses DPS\(^1\) to be cleaner and less limiting.
- Strictly *increase* the number of programs that can be written, and allow these programs to be executed and change-propagated with concurrency.

---

\(^1\)Destination-Passing Style
Current techniques work for well for divide-and-conquer-style algorithms and guarantee the following bounds:

- \( O(1) \) overhead during initial run.
- Expected \( O(\log n) \) time to propagate a single change from input to output for a variety of \( O(n \log n) \) algorithms (ie quicksort).

Current techniques are serial and assume a higher-order, functional host language (SML\(^2\)).

AFL\(^3\) is a formal description of the framework.

AFL has an API for SML.

\(^2\)SML/NJ or MLton
\(^3\)Adaptive Functional Language
Dynamic dependencies

Storing a call graph

- In order to reuse a computation, we must track the flow of data from input to output throughout the code.
- We must also track the flow of control. In particular, we need to know how to assign an order to the use of the program’s data (i.e., an order to the control nodes of the program).

Current approach: 
Acar et al

- Acar introduces the abstraction of a modref – a constrained data cell whose creation and use is tracked.
- The modref abstraction builds a dynamic dependency graph, where each use (each read) is recorded.
- Acar also uses a timestamp (order maintenance) data-structure to give total ordering and containment relationships to code blocks.
Serial initial execution

- Consumption and production of data is recorded into a dependence graph during execution ($O(1)$ overhead).
- A list of timestamps $^4$ is created that orders control nodes.
Serial change propagation

- Each control node (i.e. each use of a data cell) has a time interval which gives the nodes a total ordering and containment hierarchy.
- The total ordering is used to correctly order re-execution.
- The timestamp system is efficient and offers three necessary operations in $O(1)$ time. The corresponding problem is called order maintenance.
  - **compare** – compare two nodes for order
  - **insert** – insert a new node into the ordering at a given position.
  - **remove** – remove a given node.
Obstacles to parallelism

- The order maintenance problem is solved in terms of *total* orders – and a total ordering implies serial execution.
Obstacles to parallelism

• The order maintenance problem is solved in terms of *total* orders – and a total ordering implies serial execution.

• We really want to solve the analogous problem for *partial* orders – partial orderings allow for concurrent / independent paths of execution.
Obstacles to parallelism

- The order maintenance problem is solved in terms of *total* orders – and a total ordering implies serial execution.
- We really want to solve the analogous problem for *partial* orders – partial orderings allow for concurrent / independent paths of execution.
- But... The order maintenance problem is solved only in terms of total orders.
Obstacles to parallelism

- The order maintenance problem is solved in terms of *total* orders – and a total ordering implies serial execution.
- We really want to solve the analogous problem for *partial* orders – partial orderings allow for concurrent / independent paths of execution.
- But... The order maintenance problem is solved only in terms of total orders.
- And... Reachability in DAGs is too expensive!
Obstacles to parallelism

- The order maintenance problem is solved in terms of *total* orders – and a total ordering implies serial execution.
- We really want to solve the analogous problem for *partial* orders – partial orderings allow for concurrent / independent paths of execution.
- But... The order maintenance problem is solved only in terms of total orders.
- And... Reachability in DAGs is too expensive!

- Our solution is to constraint the DAGs to a class where the order maintenance problem can again be solved in $O(1)$ time.
Our solution

- Our solution is to constrain the ways that data can flow so that we can treat a dep graph – a DAG – as a tree.

Control-flow nodes

- `Seq E_1; E_2` – Execute `E_1` to completion, then `E_2`
- `Split \{E_1\} \{E_2\}` – Execute `E_1` and `E_2` without order restriction.

Data-use nodes

- `Create x \text{ in } E` – Create a data cell, bind it to `x`, and execute `E`.
- `Read x \text{ as } y \text{ in } E` – Read (Consume) a data cell, bind its contents to `y`, and execute `E`.
- `Write v \text{ into } x` – Write a value `v` into the initially empty data cell `x`.

- A partial order is generated for an annotated concurrent control graph – which is a tree.

- We can enforce safety statically with a special linear type system\(^5\), but only dynamically with our C API.

\(^5\) still a work in progress
Outline

1 Introduction
   Contributions
   Current approach, limitations
   Our solution

2 Program semantics

3 Change-propagation semantics

4 Summary, future work
Semantics overview (1)

Previous data-use constraints (Acar et. al.)

- Each `modref` written exactly once
- Each `modref` must be written before being read
- (Violating either of these leads to unsoundness in change propagation)

Accomplished by:

- Coupling `modref` creation and initialization into one construct
- Employing a modal type system:
  - One mode for expressions that initialize (these can also read)
  - One mode for expressions that can’t read or write `modrefs`
Semantics overview (2)

New constraints

- Same as before, plus,
- The store must be consistent in the midst of concurrent/non-deterministic operational semantics

New approach

- `modref` creation and initialization decoupled
- Unwritten `modref`s must be written exactly once
- Enforced by a sub-structural type system\(^6\)

\(^6\) eventually...
Example: list_split

fun list_split (ρ1, ρ2;
   d1:int modlist mod(ρ1)
   d2:int modlist mod(ρ2)
   l:int modlist mod(1),
   test:int → bool)

is

read l as l’ in

case l’ of

   NIL ⇒ (write NIL into d1; write NIL into d2)
| CONS(head, tail) ⇒
create d3, ρ3 in
if test head then
   write CONS(h, d3) into d1 ;
   split [ρ3, ρ2] (d3, d2, tail, test)
else
   write CONS(h, d3) into d2 ;
split [ρ1, ρ3] (d1, d3, tail, test)
end
(static) Syntax

Constr. contexts $\Delta ::= \cdot | \Delta, \eta$

Type contexts $\Gamma ::= \cdot | \Gamma, x:\tau$

Cap. contexts $C ::= \epsilon | C_1 \otimes C_2 | \eta$

Types $\tau ::= \text{int} | 1$

$\tau \text{ mod}(c)$

$\forall[\Delta](C).\tau_1 \rightarrow \tau_2$

Capabilities $c ::= \eta | 1$

Locations $\eta ::= \rho | \ell$

Values $v ::= \star | x | n$

func $f[\Delta](C;x:\tau_1):\tau_2$ is $e$ end

Expressions $e ::= v$

$v_1[C] \; v_2$

let $x$ be $e_1[C_1]$ in $e_2[C_2]$

split $e_1[C_1]$ and $e_2[C_2]$

create$\tau \; x, \rho$ in $e$

read $v$ as $x$ in $e$

write $v_1$ into $v_2$
create's, read's, write's, and seq's

\[ \Delta; C; \Gamma \vdash e : \tau \]

\[ \Delta, \rho; C \otimes \rho; \Gamma, x : \tau \text{ mod}(\rho) \vdash e : \tau' \quad \rho \notin \Delta \]

\[ \Delta; C; \Gamma \vdash \text{create}_\tau x, \rho \text{ in } e : \tau' \quad \text{(create)} \]

\[ \Delta; \Gamma \vdash v_1 : \tau \quad \Delta; \Gamma \vdash v_2 : \tau \text{ mod}(\eta) \]

\[ \Delta; \eta; \Gamma \vdash \text{write } v_1 \text{ into } v_2 : \tau \text{ mod}(1) \quad \text{(write)} \]

\[ \Delta; C_1; \Gamma \vdash e_1 : \tau_1 \]

\[ \Delta; C_2; \Gamma_{C_1}, x : \tau_1 \vdash e_2 : \tau_2 \]

\[ \Delta; C_1 \otimes C_2; \Gamma \vdash \text{let } x \text{ be } e_1[C_1] \text{ in } e_2[C_2] : \tau_2 \quad \text{(let)} \]

\[ \Delta; \Gamma \vdash v : \tau \text{ mod}(1) \quad \Delta; C; \Gamma, x : \tau \vdash e : \tau' \]

\[ \Delta; C; \Gamma \vdash \text{read } v \text{ as } x \text{ in } e : 1 \quad \text{(read)} \]
Example: quicksort

fun quicksort (ρ1;
       d1:int modlist mod(ρ1)
       l:int modlist mod(1))

is

read l as l’ in

  case l’ of
    NIL ⇒ write NIL into d1
  | CONS(pivot, tail) ⇒
    create lt, ρ2 in
    create ge, ρ3 in
    list_split [ρ2, ρ3] (lt, ge, l, (λx.x < pivot));
    create d4, ρ4 in
    split
      quicksort [ρ4] (d4, ge)
    and
    create d5, ρ5 in
    quicksort [ρ5] (d5, lt);
    append [ρ1] (d1, d5, CONS(pivot, d4)

end
split’s and functions

\[ \Delta; C; \Gamma \vdash e : \tau \]

\[ \Delta; C_1; \Gamma \vdash e_1 : \tau_1 \quad \Delta; C_2; \Gamma \vdash e_2 : \tau_2 \]

\[ \Delta; C_1 \otimes C_2; \Gamma \vdash \text{split } e_1[C_1] \text{ and } e_2[C_2] : 1 \quad \text{(split)} \]

\[ \Delta; \Gamma \vdash v_2 : \tau_1 \]

\[ \Delta; \Gamma \vdash v_1 : \forall[\Delta'] (C'). \tau_1' \rightarrow \tau_2' \]

\[ \forall[\Delta'] (C'). \tau_1' \rightarrow \tau_2' \equiv_\alpha \forall[\Delta] (C). \tau_1 \rightarrow \tau_2 \]

\[ \Delta; C; \Gamma \vdash v_1[C] \ v_2 : \tau_2 \quad \text{(app)} \]

\[ \Delta; \Gamma \vdash v : \tau \]

\[ \Delta; \Gamma \vdash \star : 1 \quad \text{(unit)} \]

\[ \Delta; \Gamma \vdash n : \text{int} \quad \text{(int)} \]

\[ \Gamma(x) = \tau \quad \Delta \vdash \tau \text{ ok} \]

\[ \Delta; \Gamma \vdash x : \tau \quad \text{(var)} \]

\[ \Delta'; C; \Gamma, f : \forall[\Delta'] (C). \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash e : \tau_2 \]

\[ \Delta; \Gamma \vdash \text{fun } f[\Delta'] (C; x : \tau_1) : \tau_2 \text{ is } e \text{ end} \quad \text{(fun)} \]

\[ \vdash \forall[\Delta'] (C). \tau_1 \rightarrow \tau_2 \]

f.21– p.26
Quicksort on 10 elements ("cluttered version")

- **Seq** nodes
- **Split** nodes
- ○ (control nodes):
  - ○ quicksort
  - ○ list_split
  - ○ list_append
- ▽ – Downtriangles are **create** nodes
- □ – Boxes are data cells (modrefs)
Quicksort on 10 elements ("less cluttered")

- **Seq nodes**
- **Split nodes**
- ○ (control nodes):
  - ○ quicksort
  - ○ list_split
  - ○ list_append
Outline

1 Introduction
   Contributions
   Current approach, limitations
   Our solution

2 Program semantics

3 Change-propagation semantics

4 Summary, future work
From a trace to a $O(1)$ partial order

- We want a way to detect independence amongst the control nodes of the trace.
- In a serial setting we had that either $A < B$ or $B < A$ (when $A \neq B$).
- Now we have three possibilities:
  - $A \prec B$, or
  - $B \prec A$, or
  - neither $A \prec B$ nor $B \prec A$,
    *i.e.*, $A$ and $B$ are independent.
- With a single total order, the last case gets disguised as one of the first two cases, creating a false dependency. To avoid this, we use *two* total orders.
From a trace to a $O(1)$ partial order

2 Topological sorts
Top-sorts $T_1$ and $T_2$ are generated alongside the trace as follows:

- For $\text{seq } A B$, both $T_1$ and $T_2$ recur in the order: $A$, then $B$.
- For $\text{split } A B$, $T_1$ recurs as $\text{seq}$, but $T_2$ recurs in reverse order.
- All other trace nodes have 0 or 1 successor and recur appropriately.

Equiv of PO ($\prec$) and 2-Top-sorts ($<_1, <_2$)

**Theorem:** The following are equivalent:

1. PO has that $A \not< B$
2. $T_1$ and $T_2$ have that $(A \not<_1 B) \lor (A \not<_2 B)$
Topological sort example

In $T_2$ we visit the children of split nodes in reverse order.
Change propagation contexts

CP contexts
A change propagation context $C$ is now a 4-tuple: $C = \langle Q_1, Q_2, t_s, t_e \rangle$,

- CP is now a recursive/forking algorithm
- $Q_i$ is a priority queue on control nodes (reads) sorted by start-times according to total order $T_i$.
- $t_s = \langle t_{s_1}, t_{s_2} \rangle$
- $t_e = \langle t_{e_1}, t_{e_2} \rangle$
- $t_s$ and $t_e$ are start and end time pairs, and $(t_{s_i}, t_{e_i})$ are intervals over total order $T_i$.
- As things re-execute, queues $Q_1$ and $Q_2$ will contain things that fall outside of their interval. These are passed to the caller as output.
Concurrent change propagation

The algorithm

1: procedure CHANGEPROP($C = \langle Q_1, Q_2, t_s, t_e \rangle$)
2:   if $C$ is exhausted then return $C$
3:   else if $Q_1 = h :: Q'_1$ and $Q_2 = h :: Q'_2$ then
4:       rerun $h$ under $C' = \langle Q'_1, Q'_2, t_s, t_e \rangle$ to get $C''$
5:       CHANGEPROP($C''$)
6:   else if $Q_1 = h_1 :: Q'_1$ and $Q_2 = h_2 :: Q'_2$ then
7:       SPLIT $C$ into $C_1$, $C_2$ and $C'$ by LCA($h_1, h_2$)
8:       Par
9:       $C'_1 = \text{CHANGEPROP}(C_1)$
10:      $C'_2 = \text{CHANGEPROP}(C_2)$
11:     EndPar
12:     $C'' = \text{MERGE} C'_1, C'_2$ and $C'$
13:     CHANGEPROP($C''$)
14:   end if
15: end procedure
Quicksort on 10 elements; then 3 added; then CP
Outline

1 Introduction
   Contributions
   Current approach, limitations
   Our solution

2 Program semantics

3 Change-propagation semantics

4 Summary, future work
Summary

Starting point: Serial framework (Acar et al)

- is strictly serial
- relies on higher-order functions (with currying, closures, etc) to use all three primitives.
- is formalized in limiting ways (destinations for writes are implicit, no support for multiple destinations).

Our extensions

- permit restricted forms of concurrency.
- are realized in C.
- have an associated formalism (still in works) which uses DPS\(^7\) to be cleaner and less limiting.
- Strictly *increase* the number of programs that can be written, and allow these programs to be executed and change-propagated with concurrency.

\(^7\)Destination-Passing Style
Future Work

(Near) future work

- Complete formal language with sub-structural typing – prove soundness
- An account of memoization and memory reuse in runtime and type system

Other future work

- An implementation of the type system and code generator
- A transformation from an ordinary functional language into CAFL
Thank you!
References