A Multithreaded Implementation for TransLucid

Toby Rahilly    John Plaice
School of Computer Science and Engineering
UNSW SYDNEY NSW 2052, Australia
{tobyr,plaice}@cse.unsw.edu.au
2008

Abstract
We present an implementation of the TransLucid language, in which expressions are evaluated in a dynamic context of arbitrary dimensionality. The basis for this implementation is the TransLucid Virtual Machine (TVM), whose execution supposes a centralised cache storing intermediate computed values and whose individual threads compute the values for (identifier, context) pairs. The TVM is designed so that it is efficiently implementable on a wide variety of physical architectures.

1 Introduction
This article presents a multithreaded implementation of the TransLucid programming language [2, 3], in which variables define hyperdatons, infinite multidimensional arrays of arbitrary dimensionality, indexed by dynamically generated lazy tuples. The infinite nature of the hyperdatons allows the natural encoding of such structures as the set of possible states in an imperative language or the set of possible functions in a functional language; it is even possible to encode hyperdatons of functions, thereby providing a simple solution to adding higher-order functions to the Lucid programming language [4]. The lazy tuples — reminiscent of those of Linda [1] — and the declarative nature of the language have been designed so that an efficient, multithreaded implementation can be generated.

Since TransLucid programs define infinite data structures, these must be sampled by making demands for (identifier, context) pairs, which will in turn provoke demands for other such pairs. For efficiency reasons, since the contexts can be of arbitrary size, possibly even infinite, only the relevant parts of the context should be examined, particularly if the intermediate computation results are cached.

The intuition for our multithreaded solution is that a new thread is launched to compute every encountered (identifier, context) pair. However, since the evaluation of the context is to be lazy, caching of intermediate values is tricky.

Therefore, when threads are launched, these will block if there is any chance that there will be recomputation. If we launch $x + y$ in a given context $\kappa$, both $x$ and $y$ will be executed in the same physical context $\kappa$, so that if the value for a dimension is required for both variables, then one of the threads will block while the other forces the value to be calculated. In general, both $x$ and $y$ will be launched before $\kappa$ is fully evaluated.

After a brief presentation of the language, we present the virtual machine, its structure and its evaluation.
The TransLucid Language

Here is the abstract syntax for expressions ($E \in \text{Expr}$):

$$
E :=
\begin{align*}
& x & | & \tau(c) \\
& \{ E < x ; E < x \} & | & \text{op}(E, \ldots, E) \\
& [ E < x ; E < x ] & | & \text{if}(E, E, E) \\
& \# E & | & x @ E \\
& \text{isspecial}(v) E & | & E ; E \\
& x ! E
\end{align*}
$$

with $x$ an identifier, $\tau$ a type and $s$ a string.

- $\tau(c)$ is a typed constants. The meaning of string $s$ may depend on the context.
- $\text{op}(E, \ldots, E)$ is a data operator.
- $\text{isspecial}(v) E$ is a special test, akin to the raising of an exception.
- $[ E < x ; E < x ]$, the relative context delta, is for building tuples relative to the context.
- $[ E < x ; E < x ]$, the absolute context delta, is for building tuples without using the context.
- $\text{if}(E, E, E)$ is the conditional operator.
- $\# E$ is the dimensional query operator, used to probe the current context.
- $x @ E$ is the context change operator, used to replace the current context.
- $E ; E$ is the explicit sequencing operator, used to force the order of side-effects, but the second operand’s answer is returned.
- $x ! E$ is the sequenced parallel operator, whose two arguments may be launched in parallel, and the second’s answer is returned.

The TransLucid Virtual Machine

The TransLucid Virtual Machine (TVM) presented in this article is the basis for our multithreaded execution of TransLucid programs. The TVM is designed so that it can be readily implemented on a wide variety of architectures, be they purely sequential, multithreaded multi-core, fully distributed, or some mix thereof. Given the current evolution of physical architectures, it is clear that architectures are becoming increasingly parallel, but that there are many diverse ways of doing so, and that heterogeneity of architectures is not likely to disappear, neither in time nor space.

The TVM supports the following features:

- a multithreaded execution model;
- a global multithreaded compatible caching system;
- an efficient lazy evaluation path;
- dynamic choice between sequential, lazy, parallel and caching evaluation models in a fine-grained manner;
- the possibility of distributed evaluations.
In the TVM, TransLucid programs are translated into bytecode, which contains information, not directly encoded in TransLucid equations, that is gathered using static analysis tasks, such as transforming dynamic typing to static typing and flattening constant expressions. Compiling to bytecode instead of native machine instructions also means that a program or part thereof can be sent across a network to another TVM, possibly crossing physical environment boundaries, such as operating systems or architectures.

The TVM evaluation engine will take as input the bytecode corresponding to a set of identifier equations, an identifier to evaluate and a context under which to evaluate. The output is the result of the evaluation.

The TVM engine begins execution by spawning one main thread for the evaluation of the original identifier-context pair demand. Should additional identifier-context pairs need to be evaluated, additional threads will be spawned, as needed, for subevaluations. This approach results in the creation of many small threads, well suited to an ideal machine with thousands of processors. During its execution, the engine uses a centralised cache storing the result of identifier-context evaluations, keeping track of only the minimal subcontexts required for these evaluations. To minimise the overhead of the caching system, the bytecode for evaluating an identifier’s definition is broken into a main instruction block and several dependent instruction blocks, which may or may not need to be executed.

The presentation style in this section is bottom-up, thereby minimising the use of forward references.

3.1 Register Memory

A register memory $M$ of size $n$ is a set of $n$ registers indexed by the register number, i.e., a mapping $M : \{0, \ldots, n - 1\} \rightarrow \mathbb{C}$. We define:

- $M(i), i = 0 \ldots n - 1$, is the value of register $i$.

For each thread $T$, three different register memories will be created:

- $M_c^T$ contains the registers with constant values.
- $M_i^T$ contains the registers used for intermediate computations when evaluating an instruction block.
- $M_d^T$ contains the registers for finalising the results of dependent computations (see below).

3.2 Instruction Block

An instruction block $I$ manipulates a register memory. It reads its arguments from registers $arg_0$, $\ldots$, $arg_{n-1}$, applies an operator thereon, and stores the result in register $store$. The instructions are:

\[
I ::= \text{Op} \left( \text{store}, n, \text{op}, arg_0, \ldots, arg_{n-1} \right) \\
\mid \text{Tuple} \left( \text{store}, n, \text{type}, arg_0, \ldots, arg_{n-1} \right) \\
\mid \text{IsSpecial} \left( \text{store}, \text{specialValue}, arg_0 \right) \\
\mid \text{Spawn} \left( \text{store}, x \right) \\
\mid \text{End} \left( \text{result} \right)
\]

- **Op**: Applies $op$ to the $n$ arguments.
- **Tuple**: Creates an $n$-tuple, relative or absolute, out of the $n$ arguments.
- **IsSpecial**: Checks if $arg_0$ is the special value $\text{specialValue}$ and stores $\text{true}$ or $\text{false}$ accordingly.
- **Spawn**: Spawns a new thread for the evaluation of $x$ under the current context.
- **End**: End of the instruction block, the result of which is stored in $\text{result}$.
3.3 Dependency Block

For each identifier, there will be a main instruction block and a dependency block, which must be satisfied before the main instruction block may be evaluated.

Dependencies encapsulate the parts of an identifier’s equation that depend on the current intensional state. Making this distinction allows us to greatly reduce the overhead of the caching system with special optimisation and static analysis used with the dependencies.

A dependency block $D$ of size $n$ is an indexed set $D : \{0, \ldots, n-1\} \rightarrow D$ of dependencies. The dependencies will be evaluated in order, unless one of them is a Conditional dependency, in which case only one of the three following dependencies (true, false, neither) will be evaluated.

The possible forms of dependency are:

\[
  d \ ::= \text{ConstantQuery}(c) \\
  \ | \text{IdentifierQuery}(x) \\
  \ | \text{Identifier}(x, l) \\
  \ | \text{Conditional}(l, n^t, n^e) \\
  \ | \text{ConditionalResult}(l, n^e) \\
  \ | \text{Sequential}(l) \\
  \ | \text{Parallel}(x)
\]

- **ConstantQuery** performs a query for the constant dimension $c$ in the current context. Places the dimension into the subcontext and the value under the dimension in the current context into the result register.
- **IdentifierQuery** spawns a new thread for the identifier $x$, with the current context. Once evaluation completes it uses the resulting value as a dimension query, placing the dimension in the subcontext and the value stored under the dimension in the current context in the result register.
- **Identifier** evaluates the instruction block to produce a tuple. A thread is spawned with the identifier $x$ and the new tuple. When evaluation completes, the result is stored in the result register.
- **Conditional** evaluates the conditional instruction block $B$ to a value $\tau(v)$. Once this value is calculated, the new dependency index becomes:
  - the current dependency index + 1, if the value is $\text{bool}(\text{true})$;
  - $n^t$, if the value is $\text{bool}(\text{false})$;
  - $n^e$, otherwise.
- **ConditionalResult** evaluates the result instruction block $B$, stores the result in the result register and moves the dependency index to the end index $n^e$.
- **Sequential** evaluates the instruction block $B$ before continuing.
- **Parallel** spawns a thread for identifier $x$ before continuing the evaluation.

3.4 Pair Memory

A pair memory is an infinite sized set of pairs indexed by memory location. A the first value of the pair can contain a value, and the second value can contain a value or a thread, i.e. a mapping $P : \{0 \mapsto (c, c \mid T)_0, \ldots, n \mapsto (c, c \mid T)_n\}$.

The virtual machine contains a single pair memory shared among all threads.
3.5 Tuple
A tuple is a set of indices into the TVM’s pair memory, $\kappa : \mathbb{N}$. We define:
- $\kappa^l \cdot \kappa^r$ is the perturbation of $\kappa^r$ onto $\kappa^l$.

3.6 Cache Entry
The cache stores information on fully and partially evaluated identifiers. Entries in a cache can be of different types and contain different parts.

\[
E := \begin{array}{c|c|c}
\text{Value} (c) & \text{Pending} \\
\text{Dimensions} (C) & \text{Undefined}
\end{array}
\]

- **Value** contains a single constant value $c$.
- **Dimensions** contains a list $C$ of dimension values that are required for the identifier’s evaluation.
- **Undefined** means this identifier is yet to be evaluated under the current subcontext.
- **Pending** means another thread of execution is already evaluating the identifier under the current subcontext.

3.7 Cache Memory
A cache memory is a set of triple tuples containing an identifier, a fully evaluated tuple and a cache entry, $S : \{(x, \kappa, S)_0, \ldots, (x, \kappa, S)_n\}$. We define:
- $S(x^l, \kappa^l)$ is the entry $S$ contained in a triple with $x^l$ and $\kappa$ such that the indices in $\kappa^l$ and $\kappa$ point to equal values in the pair memory. If no such triple exists in the cache memory, a **undefined** entry is used.

The TVM contains a single cache memory shared among all threads.

3.8 Identifier
An identifier is a list of dependencies and a instruction block. The instruction block is evaluated after the dependencies to produce the final result. Each identifier has a cache to store evaluation results.

\[
x : (D, I, U, C, n^i)
\]

- $D$ is a list of dependencies required to evaluate the identifier.
- $B$ is the instruction block evaluated after the dependencies are complete.
- $U$ is the UTF-8 string name of the identifier.
- $C$ are the constants required to evaluate $x$.
- $n^i$ is the amount of intermediate registers required to evaluate $x$. 
### 3.9 Thread

A thread is a current state of execution. At any single point in time, a thread is evaluating a identifier under a context. Threads also keep track of caching details in the subcontext. A thread can have any number of worker threads, which must complete evaluation before the parent thread can complete. Each thread also stores the state details of the current dependency and instruction to be evaluated, and the results of previous dependency calculations.

\[ T : (x, \kappa^c, \kappa^s, n^d, n^e, T, M^c, M^i, M^d) \]

- \( x \) is the identifier the thread is evaluating.
- \( \kappa^c \) is the context under which the identifier is evaluated.
- \( \kappa^s \) is the subcontext, dimensions of the context required to reach the current \( I^d \).
- \( n^d \) is the index of the dependency on \( X \) to be evaluated.
- \( n^e \) is the index of the instruction contained in a instruction block being evaluated.
- \( T \supset \{(n^d_1, T_1), \ldots, (n^d_n, T_n)\} \) is a set of pairs that contain a thread that has been spawned for a dependency evaluation, and the dependency’s index.
- \( M^c \) is the register memory contains constants required for instruction block evaluations. The registers contain a copy of the constants in \( C \) of \( x \).
- \( M^i \) is the register memory used during instruction block evaluations. The memory size is \( n^i \) from \( x \).
- \( M^d \) is the register memory in which dependency results are stored.

### 3.10 TransLucid Virtual Machine

The TransLucid Virtual Machine consists of a thread tree, where each node is a thread and its children are the threads spawned by the node thread. The root thread is the thread evaluating the initial demand.

The TVM also contains a pair memory, a cache memory and the set of identifiers in the system. Once the root thread is completed evaluation, then its result is the final result of the evaluation is complete.

\[ V : (T, X, S, P) \]

- \( T \) is the main thread of execution.
- \( X \) is a list of identifiers loaded into the machine.
- \( S \) is a cache memory.
- \( P \) is a pair memory.

### 4 Multithreaded Evaluation

As described in §3 threads are spawned in the TVM when a tuple is created and when an identifier dependency is met. The difficulty comes in ensuring that no two threads calculate an identifier under the same subcontext, i.e. there is no redundant evaluation between different threads of execution.
4.1 Sequential Cache

The minimal evaluation is ensured by using a cache that stores identifier evaluation results under fully evaluated contexts, and also the set of dimensions required to evaluate the identifier.

Before evaluating an identifier’s instruction blocks under a context, the cache is checked for a result for that identifier under the empty subcontext. If a result is stored, it takes the result, if dimensions are stored it copies the range of dimension value pairs from the context to the subcontext and searches the cache again. If the cache is found to be empty under the subcontext, then the identifier is evaluated under the current context with the subcontext built from the warehouse search.

When evaluation requires a dimension query, if the dimension is not already stored in the subcontext, the dimension is stored in the cache under the subcontext, and then added to the subcontext.

Once evaluation is complete, the result is added to the cache.

4.2 Multithreaded Cache

When two threads are evaluating different branches of a problem, we want to ensure that they do not both evaluate the same branch. We do not want two threads to calculate a specific identifier under the same subcontext.

We achieve minimal evaluation by using a shared cache between the threads that meets the following properties:

- Only one thread may write to a single cache entry at a time;
- Threads should not read a cache entry while it is being written to;
- To ensure the minimum evaluation property, only one thread should be able to evaluate an empty cache entry at a time.

The first two properties imply that the implementation needs to ensure that accesses to entries in the cache are mutually exclusive. The mutual exclusion should be fine-grained, each cache entry being locked separately. This exclusion could be implemented with lock synchronization, or using a transactional memory.

However the third property means that a thread should ‘claim’ a cache entry it is about to evaluate. Any other thread that comes to a claimed cache entry should wait for the thread that claimed the entry to fill the cache entry and unclaim it before taking the entry’s value.

As an example consider two threads attempting to evaluate the TransLucid equation ‘id = #0;;’ concurrently. The first thread (A) will try and access the identifier ‘id’ under the empty subcontext, and will see that it is empty, so will claim it for evaluation. When the second thread (B) tries to access the same cache entry, it will see that the entry has been claimed, so it will wait on the entry. Thread A will continue to evaluate ‘id’. When thread A comes across the query for the ‘0’ dimension, it will store a demand in the cache under the entry ‘id’ with the empty subcontext, which it had previously claimed. Thread A will then claim the cache entry of ‘id’ with the new subcontext of thread A which now contains the ‘0’ dimension. Finally thread A unclaims the ‘id’ with an empty context that thread B is waiting on. Now that thread B has been released, it will now take the stored dimension demand ‘0’, place it in its subcontext and load the cache entry under ‘id’ and the new subcontext of thread B. If threads A and B have the same value under dimension ‘0’, then thread B will wait for the claimed entry again, otherwise it will now be in a different context space to thread A, and will claim the entry to itself and evaluate.

There is however a problem in the logic, that causes the concurrent evaluation to be less efficient than possible. Currently when a following thread is waiting for a thread to complete its evaluation of a warehouse entry, it must wait for it to complete evaluation of all the identifier’s subidentifiers. When an entry is added to the cache of a sub identifier, a following thread that is waiting on the parent identifiers cache entry will not be notified, and hence kept waiting. What
this means is that often a following identifier will have to wait for the leading thread to evaluate an entire identifier, even if the subcontexts are quite different.

The solution to this problem is to allow following threads to evaluate the dependencies of the identifier, but not the instruction blocks. Before each instruction block, the following thread checks the warehouse for a new entry, if there is one then it must still be following and does evaluate the instruction block, instead taking the value from the cache. This allows threads to separate their evaluation paths earlier, and also does not allow for duplicate evaluation of instruction blocks.

4.3 Cache Rules

The cache stores past identifier evaluations. Before a thread is executed, the cache is checked to see if it contains any information about the threads state.

4.3.1 Look Up

\[
V_0, \kappa^c, \kappa^s_0 \vdash x : V_1, \kappa^s_1, c
\]

- \( V \) is the TVM in which the evaluation is occurring.
- \( \kappa^c \) is the context under which to evaluate \( x \).
- \( \kappa^s_0 \) is the current subcontext.
- \( x \) is the identifier to evaluate.
- \( c \) is the result.
- \( \kappa^s_1 \) is the subcontext used to find \( c \).

If nothing is found in the cache under the subcontext, the thread needs to be executed.

\[
V_0 = (T, X, S, P)
S(x, \kappa^s_0) = Undef\text{in}ed
V_2 = (T, X, S \cup \{(x, \kappa^s_0, \text{Pending})\}, P)
\]

\[
V_2, \kappa^c, \kappa^s_0 \vdash \quad x : V_3, \kappa^s_1, c
\]

\[
V_0, \kappa^c, \kappa^s_0 \vdash \quad x : V_4, \kappa^s_1, c
\]

If dimensions are found in the cache under the subcontext, add the dimensions to the subcontext and search again.

\[
V_0 = (T, X, S, P)
S(x, \kappa^s_0) = \text{Dimensions} \{(c^d_1, \ldots, c^d_n)\}
V_{i-1} \vdash \kappa^c(c^d_i) : V_i, c^v_i
\]

\[
\kappa^s_1 = \kappa^s_0 \cup \{c^d_1 \mapsto c^v_1, \ldots, c^d_n \mapsto c^v_n\}
\]

\[
V_{i\text{, }i-1}, \kappa^c, \kappa^s_1 \vdash \quad x : V_{i\text{, }i-1}, \kappa^s_2, c
\]

\[
V_0, \kappa^c, \kappa^s_0 \vdash \quad x : V_{i\text{, }i-1}, \kappa^s_2, c
\]

If a result is found, return that value with the subcontext.

\[
V = (T, X, S, P)
S(x, \kappa^s) = \text{Value} (c)
\]

\[
V, \kappa^c, \kappa^s \vdash \quad x : V, \kappa^c, c
\]
If a pending entry is found, the identifier is evaluated in the following mode, which does not check the cache for updated entries for evaluating instruction blocks.

<table>
<thead>
<tr>
<th>$V_0$</th>
<th>$(T, X, S, P)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S(x, \kappa_0^s)$</td>
<td>$\text{Pending}$</td>
</tr>
<tr>
<td>$V_0, \kappa^c, \kappa_0^s \vdash x : V_1, \kappa_1^s, c$</td>
<td></td>
</tr>
<tr>
<td>$V_0, \kappa^c, \kappa^s \vdash x : V_1, \kappa_1^s, c$</td>
<td></td>
</tr>
</tbody>
</table>

### 4.3.2 Store Dimension Entry

Stores a list of dimension values under the subcontext into the cache. The next entry to be evaluated is found by placing the dimensions into the subcontext. The next entry can then be set to $\text{Pending}$.

<table>
<thead>
<tr>
<th>$V_0$</th>
<th>$(T, X, S_0, P)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\kappa_1^s$</td>
<td>$\kappa_0^s \cdot (\kappa^c(C))$</td>
</tr>
<tr>
<td>$S_1$</td>
<td>$S_0 \cup {(x, \kappa_0^s, \text{Dimensions}(C))}$</td>
</tr>
<tr>
<td>$S_2$</td>
<td>$S_1 \cup {(x, \kappa_1^s, \text{Pending})}$</td>
</tr>
<tr>
<td>$V_1$</td>
<td>$(T, X, S_2, P)$</td>
</tr>
</tbody>
</table>

- $V$ is the list of dimensions to store.
- $\kappa_0^s$ is the subcontext under which to store the dimensions.
- $\kappa_1^s$ is the subcontext after merging with the dimension list, and under which the cache entry is set to $\text{Pending}$.

### 4.3.3 Store Value Entry

Stores a value under the subcontext into the cache.

<table>
<thead>
<tr>
<th>$V_0$</th>
<th>$(T, X, S_0, P)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>$S_0 \cup {(x, \kappa^s, \text{Value}(c))}$</td>
</tr>
<tr>
<td>$V_1$</td>
<td>$(T, X, S_1, P)$</td>
</tr>
</tbody>
</table>

- $c$ is the value to store under $x$ and $\kappa^s$ in the cache.

### 5 Conclusion

The TransLucid Virtual Machine has been fully implemented and is well suited to the current range of chips with a limited number of cores, because of the combination of centralised cache and of independently running threads. The cache effectively acts as a large multidimensional transactional memory, which works well when there is little clash for the individual cells.

On purely sequential problems, such as Ackermann’s function, additional cores do not lead to any speedup. On problems with limited branching, such as Takeuchi’s function, some speedup does occur. On problems that naturally branch out, such as divide and conquer problems, we have achieved a speedup of 1.6 on two processors, i.e., an efficiency of 80%. These results are encouraging, and we believe that this efficiency can be improved by reducing memory contention between threads.
References


