Dynamic Program Parallelization

Lorenz Huelsbergen and James R. Larus*
University of Wisconsin-Madison†

Abstract

Static program analysis limits the performance improvements possible from compile-time parallelization. *Dynamic program parallelization* shifts a portion of the analysis from compile-time to run-time, thereby enabling optimizations whose static detection is overly expensive or impossible. *Lambda tagging* and *heap resolution* are two new techniques for finding loop and non-loop parallelism in imperative, sequential languages with first-class procedures and destructive heap operations (*e.g.*, ML and Scheme).

Lambda tagging annotates procedures during compilation with a tag that describes the side effects that a procedure's application may cause. During program execution, the program refines and examines tags to identify computations that may safely execute in parallel. Heap resolution uses reference counts to dynamically detect potential heap aliases and to coordinate parallel access to shared structures. An implementation of lambda tagging and heap resolution in an optimizing ML compiler for a shared-memory parallel computer demonstrates that the overhead incurred by these run-time methods is easily offset by dynamically-exposed parallelism and that non-trivial procedures can be automatically parallelized with these techniques.

*This work was supported in part by the National Science Foundation under grant CCR-9101035 and by the Wisconsin Alumni Research Foundation.

†Authors' address: Department of Computer Sciences, 1210 West Dayton Street, Madison, Wisconsin 53706. Email: {lorenz,larus}@cs.wisc.edu

1992 ACM LISP & F.P.-6/92/CA
© 1992 ACM 0-89791-483-X/92/0006/0311...$1.50

1 Introduction

Implicit parallelization of programs written in sequential programming languages is attractive—it eases programming, allows effective maintenance of large systems, and extends a program's portability across parallel and sequential architectures. Existing techniques for parallelizing imperative languages are primarily static—in that the analysis, optimization, and parallelization is performed at compile-time. Static techniques for extracting parallelism from sequential programs are inadequate. Abstract interpretation [6], data flow analysis [3, 8], and methods based on inferencing [12] conservatively approximate a program's dynamic behavior and typically underutilize the program's potential parallelism in order to preserve its sequential semantics. In addition to being conservative, static techniques are costly since they must be applied interprocedurally on the entire text of a program.

This paper proposes *dynamic* parallelization of "mostly functional" languages such as ML[14], Scheme[17], and Lisp that allow side effects, input/output, and higher-order procedures.[1] Dynamic parallelization is a *hybrid*, composed of static and dynamic components. Fast static analysis provides partial information during compilation. Other information, which is difficult and inefficient to collect statically, is gathered at run-time when it is often readily available. Dynamic program parallelization supplements, not supplants, existing analyses by extending them to expressive language features, such as first-class procedures and heap side effects, and by reducing their cost and simplifying their implementations by only computing partial information at compile-time.

We have developed and tested two dynamic parallelization techniques. *Lambda tagging* (λ-tagging)

[1] Procedures that invoke abnormal control flow (*e.g.*, explicit continuations) are currently not handled by these techniques—this does not, however, preclude parallelization of *other* procedures within the same program.

```
fun map _ [] = []
  | map f (a::x) = (f a) :: (map f x)
```

(a)

```
fun mapParallel _ [] = []
  | mapParallel f (a::x) =
    let val (y,z) = PCALL(f a,mapParallel f x) in
        y::z
    end
```

(b)

Figure 1: Sequential (a) and parallel (b) versions of `map`. Static analyses must select version (a) if `f` cannot be determined to be side-effect free. The PCALL [5] construct forks its arguments as parallel threads and tuples their results when the threads join.

discovers parallel threads in the presence of higher-order procedures. The second technique, *heap resolution*, permits threads to concurrently modify non-overlapping heap structures. Both techniques uncover loop and non-loop parallelism. We have implemented λ-tagging and heap resolution in the Standard ML of New Jersey (SML/NJ) optimizing compiler [1] on a Sequent shared-memory parallel computer. Timings of programs hand-annotated with λ-tags and heap resolution information indicate that significant speedup can be obtained with these dynamic techniques and an efficient compiler.

The viability of dynamic parallelization hinges on efficient run-time components that introduce little overhead into a program's execution. Our efforts to-date have been directed towards designing and implementing efficient run-time components. The static aspects of λ-tagging and heap resolution (analysis and restructuring) were performed manually. The experiments in this paper clearly demonstrate the practicality of the approach, and consequently, we are proceeding with the compiler.

Following an overview of λ-tagging and heap resolution, the paper describes the static and dynamic components required for λ-tagging (Section 2) and heap resolution (Section 3). Section 4 describes our implementation and presents empirical results of λ-tagging and heap resolution. Related work is discussed in Section 5.

1.1 Lambda Tagging Overview

Higher-order languages with first-class procedures pose difficulties for static analyses since an individual procedure call site can invoke many procedures and the task of determining the set of procedures invoked from a call site is difficult [16, 18]. Static parallelization systems, therefore, do not precisely analyze programs involving first-class procedures [6, 10]. The ML procedure `map`[2] (Figure 1a) illustrates the problem and serves as an example for dynamic parallelization with λ-tags. A call to `map` may safely use a parallel version of `map` (Figure 1b) if application of the procedure parameter `f` does not exhibit side effects. If the identity or behavior of a procedure passed at a call site cannot be safely deduced, static methods must err conservatively and use sequential code. Even when the procedures invoked at a call site are known, compiler analyses statically approximate multiple procedures reaching the site as having the effect of the most destructive procedure, although an actual call may invoke a side-effect-free procedure.

λ-tagging annotates procedures with a description of their potential side-effects. A procedure's λ-tag is assigned at compile-time when possible. Otherwise, a λ-tag is constructed at run-time when the procedure's closure is formed. λ-tags are dynamically propagated with procedures' run-time representations (closures). Statically-inserted checks examine λ-tags at run-time to determine when parallel evaluation is safe (preserves the sequential semantics). λ-tags allow dynamic detection of parallel computations involving dynamically-created procedures, as well as procedures propagated through data structures too complex for static analysis.

λ-tagging sidesteps the problem of not knowing a procedure's side effects at compile-time by examining λ-tags on higher-order procedures at run-time. Figure 2 is a restructured version of `map` containing both sequential and parallel versions. On entry to `map`, the effect of the higher-order parameter `f` (described by `f`'s λ-tag) is used to select the parallel or sequential version of `map`. Iterations of `map` may evaluate in parallel if concurrent instantiations of `f` cannot interfere. The call site from which the higher-order procedure originated is irrelevant. In addition, the cost of checking a procedure's tag is small and is only incurred once upon entry to `map`.

[2] `map` is a canonical example of potential parallel evaluation obscured by unknown side effects of higher-order procedure parameters. Other common examples are procedures traversing recursive data structures and sorting algorithms parameterized by a comparison predicate.

```
fun map _ [] = []
  | map f l =
    let fun mapSequential [] = []
          | mapSequential (a::x) =
            (f a) :: (mapSequential x)
        fun mapParallel [] = []
          | mapParallel (a::x) =
            let val (y,z) = PCALL(f a,mapParallel x)
            in
                y::z
            end
    in
        (* select map based on f's λ-tag *)
        if SAFE f then mapParallel l
        else mapSequential l
    end
```

Figure 2: Transformed version of `map` that dynamically selects sequential or parallel evaluation depending on the effects of the procedure bound to `f`. The primitive predicate SAFE examines a run-time λ-tag from the procedure bound to `f`.

1.2 Heap Resolution Overview

Heap resolution dynamically detects and resolves potential heap side-effect conflicts. At run-time, the exact shape of dynamic data structures is known. Since a structure's topology is often determined by program input unavailable at compile-time, this problem is ideally suited to solution by dynamic parallelization. Existing static analyses provide crude, yet expensive, approximations to dynamic structures. These compile-time parallelization techniques are often forced to assume structure sharing due to imprecise alias information [7, 6, 10]. Heap resolution is based on the observation that heap reference counts identify sharing in a heap structure.

The destructive quicksort `qs` of Figure 3 serves as the example for heap resolution. `qs` sorts the elements of list `l` according to a comparison predicate `cmp`. This version of quicksort partitions `l` in place. If elements of `l` do not share structure, the arguments to (destructive) **append** are disjoint and may evaluate concurrently. If elements of `l` share structure, parallel evaluation of these arguments must dynamically coordinate access to shared structure. It is difficult (or impossible) for a compiler or programmer to detect that the arguments to **append** are disjoint.

Heap resolution is applicable to heap structures with acyclic spines, *e.g.* a non-circular list of arbitrary graphs. In the above example, a programmer has declared the datatypes **pair** and **rlist** as acyclic. Note that individual elements of the **rlist** being sorted may contain cyclic structures.

The static component of heap resolution identifies expressions that modify the heap, but may execute in parallel when the side effects are to disjoint portions of the heap. These expressions are statically scheduled for parallel evaluation, but are altered to dynamically examine a heap node's reference count before accessing it. A *linearization* [20] of parallel threads preserves the language's sequential semantics by coordinating accesses to, and modifications of, potentially shared nodes. A thread consults this linearization to determine whether it may access a heap node that is potentially shared or whether it must suspend until prior threads in the linearization complete.

In the example, `(qs left cmp)` and `(qs right cmp)` are statically selected for parallel evaluation using heap resolution. The declaration that a **pair** is acyclic implies that the partitioned sublists, `left` and `right`, can reach a common heap node h only if all paths from `left` to h (and `right` to h) contain a node with reference count greater than one. `(qs right cmp)` may evaluate in parallel with `(qs left cmp)` providing that evaluation of `(qs right cmp)` suspends upon access to a heap node with reference count greater than one. If elements of the list `l` being sorted are not shared, the sort is completely parallel. Shared elements inhibit parallelism, but portions of the sort may still execute concurrently.

2 Lambda Tagging

λ-tagging statically annotates procedures at compile-time with an approximation to their potential side effects. These λ-tags are used at run-time to build consistent λ-tags for dynamically created procedures and to make parallelization decisions.

2.1 Static λ-tagging

The static component of λ-tagging determines the side effects potentially exhibited by a program's procedures. Procedures whose effects cannot be fully determined at compile-time are statically restructured to compute these tags dynamically. Statically, checks are generated to select parallel or sequential evaluation based on λ-tag information.

2.1.1 Describing Side Effects

Side-effects are described using FX-like effect descriptions [12]. An ML expression may perform input or output, read or write data structures in the heap, or

```
acyclic datatype 'a rlist = rNil | rList of ('a * 'a rlist ref)
acyclic datatype 'a pair = Pair of ('a * 'a)

fun qs rNil _ = rNil
  | qs (rList(a,x as ref x')) cmp =
    let fun split _ rNil = Pair(rNil,rNil)
          | split pivot l =
            let fun split' rNil less greater = Pair(less,greater)
                  | split' (l as rList(a,x as ref x')) less greater =
                    if cmp pivot a then (x := less; split' x' l greater)
                    else (x := greater; split' x' less l)
            in
               split' l rNil rNil
            end
        val _ = x := rNil
        val Pair(left,right) = split a x'
    in
       append (qs left cmp) (rList(a,ref (qs right cmp)))
    end
```

Figure 3: Destructive quicksort procedure. Heap resolution evaluates the arguments to (destructive) append in parallel. The declaration *acyclic* on the mutable `'a rlist` datatype indicates that no part of the spine of the list participates in a cycle (elements of the list may, however, contain cycles).

have no visible side effects. These primitive side effects are denoted i/o, read, and write.[3] An expression without side effects has effect pure. The effect of an expression that performs multiple primitive operators is a composite effect. A composite effect is an effect, $\epsilon = \epsilon_1 \sqcup \epsilon_2$, where ϵ_1 and ϵ_2 are effects. Effects induce a finite lattice, with bottom element pure and top element i/o⊔read⊔write.

An ML expression that modifies the contents of a `ref` cell with the `:=` operator produces a write in the heap. This write potentially interferes with expressions that read the heap (dereference pointers with `!` or pattern match `ref` types) or may conflict with other expressions that also write the heap. The ML `print` operator produces an i/o effect. Expressions that do not have conflicting effects are candidates for parallel evaluation, whereas expressions with potentially conflicting effects must evaluate sequentially.

2.1.2 Static Computation of Effect Tags

Static effect tag assignment determines which procedures of the program are purely functional (have no side effects), potentially functional (the procedures depend on the effect of higher-order procedures or apply procedures whose effect cannot be statically determined), or destructive (modify extant structures in the heap or perform I/O). The effect of a procedure p consists of the maximum effect of expressions in p, including procedures invoked by p. Effects of procedures invoked by p are either determined statically or computed dynamically.

The *base effect* of a procedure p is the portion of p's effect that can be determined statically. If p's entire effect can be determined statically, p's run-time λ-tag simply carries p's base effect. The base effect is computed in a manner to similar to the effect computation of FX [12]. FX conservatively approximates the effect of procedures whose effect depends on statically-unknown procedures. Our analysis instead identifies procedures invoked by p whose effects are statically unknown. Their effect is incorporated into procedure p's effect at run-time rather than approximated at compile-time. Statically, p has an incomplete effect. In other respects, our computation of p's base effect is identical to effect inferencing in FX. The effect of procedure p is the least upper bound of the effects of p's constituent sub-expressions.

Procedure p has a *parametric effect* if p's effect depends on the effect of other procedures whose effects are unknown at compile-time. A parametric effect may represent the (yet unknown) effect of free procedures in p or higher-order parameters to p. Parametric ef-

[3] The FX alloc effect is not included here. It is assumed that parallel threads allocate storage from distinct sections of the heap. Allocation, therefore, cannot cause conflict. FX effects are also parameterized by a *region* describing where the effect may occur. For our purposes, read and write effects can occur anywhere in heap and a region description is unnecessary.

fects (if any) in conjunction with the base effect form a procedure's total effect. A procedure p that has a parametric effect initially carries a λ-tag denoting the maximum effect (i/o⊔read⊔write). This initial, conservative effect tag is necessary since p may be used as a higher-order procedure, in which case the origin of p's parametric effect component is yet unknown. A parametric effect due to higher-order parameters to p dynamically selects a specialized (*e.g.*, parallel or sequential) version of p (§2.2). Procedure p is dynamically re-tagged if components of p's parametric effect are due to free procedures in p (§2.3).

If procedure p invokes a procedure f that is free in p and f's effect is not statically available, p's full effect can be computed dynamically when a closure for p is formed. At that point, a λ-tag for p can be dynamically computed from f's effect (carried by f's λ-tag) and p's base effect.

Procedure p's effect may also depend on the effect of higher-order procedure parameters to p. In this case, the effect of these parameters (as indicated by their λ-tags) can be dynamically examined when p is invoked and an appropriate version of p executed (see Figure 2). ML's static type checker [14] provides information sufficient to determine which formal parameters to a procedure p represent higher-order procedures possibly invoked by p. For example, the type signature for procedure `map`, `map:('a -> 'b) -> 'a list -> 'b list`, indicates that `map`'s first parameter is a higher-order procedure.

Finally, it is possible that procedure p's effect becomes apparent only during p's execution. This occurs when p acquires and applies an unknown procedure during its execution. For example, p might retrieve and apply an unknown procedure from a data structure. At compile-time, p is conservatively assigned a λ-tag with the maximum effect.

The procedure `map` (Figure 1) has a base effect of pure since the list constructor `::` has effect pure. `map` also has a parametric effect since the higher-order parameter to `map` has unknown effect. Therefore, the λ-tag assigned to `map` corresponds to the maximum effect, for if `map` itself is passed as a higher-order parameter, its eventual parameters (and hence its effect) are unknown.

The algorithm to compute a procedure p's static effect first determines the types of p and expressions in p. This information identifies unknown procedures potentially invoked by p. The effect of these unknown procedures forms p's parametric effect component (supplied at run-time). The base effect of p is then computed using the FX effect-inferencing algorithms [9, 12]. Effect information for p is used to assign p's initial λ-tag (as previously described) and to restructure p (§2.2).

2.1.3 Time Complexity

Detection of higher-order parameters to a procedure p and free procedures in p requires a type signature for p and the types of the expressions in p. The ML type checker [14, 13] provides this information. Effect reconstruction for a language with ML-style polymorphism requires a polynomial-time algorithm [9]. We believe that we can compute static effect tags in at most polynomial time since our effect system does not need to statically approximate the effect of unknown procedures. The effect of an unknown procedure is merely noted as being available at run-time.

Static effect determination is tractable, even for large programs, since the static analysis is applied to each procedure separately. The time complexity is a function of the size of a procedure—it is not a function of the size of the entire program.

2.2 Using Effect Information

A procedure's static effect information, its base and parametric effects, allows restructuring of the program to dynamically examine effect λ-tags. Two types of restructuring are necessary: creation of λ-tags for dynamic procedures and inspection of λ-tags for parallelization. If a procedure p has a parametric effect due to free procedures in p, the code that forms the closure for p is restructured to incorporate the effects of p's free procedures into p's run-time λ-tag. Dynamic effect combination is described in Section 2.3.

Procedures with parametric effects due to higher-order parameters present opportunities for parallelization. A procedure p whose effect is dependent on a higher-order parameter f can be compiled into multiple versions, each optimized for a particular effect (or set of effects) of f (*cf.* Chambers & Ungar's SELF compiler [2]). For all possible effects ϵ, if expressions in p may safely evaluate in parallel given that f has effect ϵ, a new version of p, p_ϵ is built. A check to select p_ϵ when f has effect ϵ is inserted into p. For example, the following dynamic version of procedure `map` (Figure 1) is created by propagating possible effects of `f` into `map` (and merging identical versions):

```
fun map _ [] = []
  | map f (a::x) =
    if SAFE f then
        let val (x,y) = PCALL(f a,map f x) in
            x::y
        end
    else (f a)::(map f x)
```

When a pure (or read) effect is propagated into `map` as `f`'s effect, the parallel code in the consequent of the `if` is generated. The original (sequential) code for `map` is retained when `f`'s effect contains a write or i/o. This

naive restructured version of `map` can be further optimized to create the efficient dynamic `map` of Figure 2 by recognizing that `f`'s effect is loop invariant.

If the procedure p being restructured receives k higher-order parameters, all possible effect combinations for these parameters can be propagated into p and p restructured accordingly. This is viable if k and the effect lattice are small. Alternately, for large k or a large effect lattice, a bottom-up approach can be used that examines p's sub-expressions and determines the higher-order parameters of p for which a safety check would lead to parallelization of these sub-expressions. Appropriate versions of p and a corresponding check are then generated.

2.3 Dynamic Computation of λ-tags

At run-time, a procedure's effect is propagated with the procedure's representation (closure) in the λ-tag. Furthermore, new λ-tags are computed for dynamic procedures whose effect was not available at compile-time. When the closure for such a procedure p is dynamically created, its effect is computed from other procedures' λ-tags and p's current λ-tag. Procedure p is re-tagged with this updated effect. The static restructuring phase identifies these closures and notes the procedures required to compute the new λ-tag. As an example, the inner anonymous procedure in the ML procedure

```
fun compose f = (fn g => fn x => f (g x))
```

to compose two procedures is dynamically assigned a λ-tag consistent with the composition of the effects of `f` and `g` (as indicated by their λ-tags) when a closure for `(fn x => f (g x))` is formed. This dynamically created λ-tag is examined at run-time in the same manner as statically assigned tags. Statically, `(fn x => f (g x))` has a parametric effect because of the unknown free procedures `f` and `g`. Due to this parametric effect, the procedure `(fn x => f (g x))` is (conceptually) compiled to:

```
SET (fn x => f (g x)) (COMBINE (GET f) (GET g))
```

The procedure GET retrieves a procedure's λ-tag. SET sets the λ-tag of its first parameter (a closure) to the value of its second parameter and returns the closure. COMBINE computes the maximum effect (least upper bound) of two λ-tags. λ-tags are combined using the effect lattice of Section 2.1.1. To combine effects efficiently, a run-time effect representation that admits a fast, least upper-bound operator is used.[4]

[4]For our effect lattice, small bit vectors manipulated with addition and bit-wise logical operations suffice.

3 Heap Resolution

Heap resolution, our second dynamic parallelization technique, allows concurrent modification of heap structures and is orthogonal to λ-tagging. Heap resolution orders conflicting heap accesses while permitting parallel threads to execute concurrently. In particular, heap resolution permits structure traversals that modify non-overlapping sets of objects to proceed in parallel. These traversals can be expressed as recursive loops with side-effecting bodies or as *non-linear* recursion [10] over arbitrary structures.

Parallel evaluation of expressions that destructively access shared data must prevent read/write and write/write conflicts from altering the sequential semantics of a program. Detecting and synchronizing data races in dynamic shared data is difficult for compilers and programmers since sharing appears (and disappears) dynamically and is often dependent on program input. However, at run-time, shared data can be detected and access to it correctly coordinated. For example, a compiler may deduce that a list l of heap elements might contain the same element more than once (thereby sharing it with itself) and force access to l to be sequential. For a given execution of the program, however, l's elements may be disjoint so that parallel access and modification of them is safe. Even if some elements of l are identical (shared), others can be processed in parallel if sharing is detected dynamically.

To preserve the sequential semantics of a program, control-independent[5] expressions evaluating in parallel must see changes to the program's state in the order produced by a sequential schedule of the expressions. Let e_1 and e_2 be control-independent expressions with a sequential schedule that evaluates e_1 before e_2. Semantics-preserving parallel evaluation of e_1 and e_2 must allow e_1 to modify all state visible by e_2 before e_2 accesses this state. Furthermore, e_2 must not modify state shared with e_1 until e_1 has ceased accessing it. Heap resolution prevents e_2 from modifying or accessing heap structures shared with e_1 until e_1 completes. If e_1 and e_2 do not share data, both expressions evaluate in parallel. Otherwise, evaluation of e_1 and e_2 is parallel until e_2 attempts to access potentially-shared data, at which point evaluation of e_2 suspends until e_1 completes.

Heap resolution requires information about which heap structures are shared (accessible via multiple

[5]Expressions $e_1, \ldots, e_n$ are *control independent* if the evaluation of $e_1, \ldots, e_n$ is constrained only by data dependences. Examples of control-independent expressions include arguments in a procedure application, bodies (and bindings) in `let` clauses, and sequential statement lists.

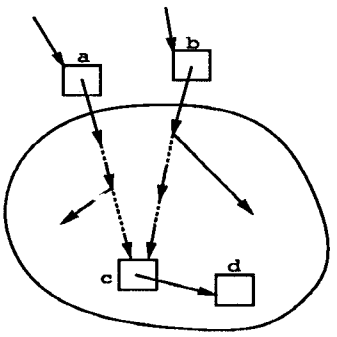


Figure 4: A sample heap. If no paths from a to b or from b to a exist, a thread t_1 with access to a can detect an access to heap nodes potentially shared with a parallel thread t_2 that has access to b, by examining node reference counts. The reference count greater than one on node c indicates potentially-shared data (c and d).

paths in the heap) and could be concurrently referenced by multiple threads. Reference counts provide a cheap and effective approximation to this information. The following definitions are useful for describing when reference counts can be used to detect shared heap nodes. Let $\mathcal{H}$ denote the heap and h and h' nodes in $\mathcal{H}$.

Definition 1 A node $h \in \mathcal{H}$ *reaches* node $h' \in \mathcal{H}$ if a path from h to h' in $\mathcal{H}$ exists. A node h *reaches* itself if a non-zero length path from h to h exists.

Definition 2 A *popular* node $h \in \mathcal{H}$ is a node that is directly referenced by two or more nodes in $\mathcal{H}$.

We can now state the property that forms the basis of heap resolution. Suppose heap node h does not reach heap node h' and h' does not reach h. If h and h' both reach a common node n, then all paths from h to n (and h' to n) must contain a popular node. A popular node is an indicator of sharing. Not all nodes shared by h and h' are popular, but these can only be reached through a popular node. A heap node's dynamic reference count indicates its popularity.

The heap in Figure 4 provides an illustration. Nodes a and b cannot reach one another. Nodes a and b, however, reach common nodes (c, and node d reachable from c). Parallel threads, one accessing a and the other accessing b, dynamically detect the potential reference to a shared heap node when a popular node (a node with reference count greater than one) is encountered.

Heap resolution relies on static analysis to select expressions for parallel evaluation and to determine static relations among variables pointing into the heap. This static analysis, in conjunction with programmer declaration of structures known to be acyclic, provides information about the relationship of some heap nodes at compile-time. We now describe the run-time support required for heap resolution (§3.1) and the programmer datatype declaration (§3.2). Section 3.3 describes a simple static analysis that is sufficiently powerful to automatically parallelize a large class of expressions. Markers, introduced in Section 3.4, augment reference counts in a case where the simpler mechanism does not suffice.

3.1 Operation of Heap Resolution

At run-time, heap resolution coordinates access to potentially-shared heap nodes. The compiler produces two versions of all procedures that can be invoked by expressions evaluated with heap resolution (§3.3). Let p be such a procedure. In addition to the original version of p, the second version, $\overline{p}$, detects and arbitrates access to potentially-shared heap nodes. The original version of p executes during sequential execution of the program. $\overline{p}$ is called in lieu of p when parallel threads with potential side effects are scheduled. $\overline{p}$ examines reference counts to detect popular nodes on heap accesses. As noted above, popular nodes delimit potentially-shared structure, which must be accessed sequentially to preserve the intended semantics.

3.1.1 Reference Counts

Heap resolution uses reference counts to detect dynamic sharing of heap nodes. Reference counts are maintained only for pointers from heap nodes to other heap nodes. References from variables pointing at heap nodes are not counted since they do not expose information about the heap's topology. Reference counts must be maintained during the entire execution of a program, but are examined only when expressions evaluate in parallel using heap resolution.

Reference counts are incremented when new structure is built or when existing pointers in the heap are reassigned. A heap node's reference count is decremented when a heap pointer to it is removed. While parallel threads are evaluating with heap resolution, a popular node may not be made unpopular, *i.e.*, a node's reference count may not be lowered from two to one. This restriction is necessary since a thread may maintain local pointers in variables which do not affect reference counts. If a thread t has such a local pointer to a popular node h and t makes h unpopular, t may inadvertently grant a concurrent thread access to h (which is now accessible by t through a pointer variable and by some other heap node). Decrementing the reference count of h must be delayed until all parallel threads complete. This can be implemented

by maintaining a list of nodes to be decremented and decrementing the reference counts of these nodes when sequential evaluation resumes. We expect this situation to occur infrequently.

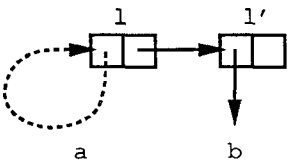
Reference counts provide a conservative estimate of sharing and can therefore indicate *false sharing* (*e.g.*, sharing due to structure not accessed by parallel threads, or to discarded structure not yet removed by the garbage collector). The safe approximation they provide is, however, less conservative than static techniques that approximate all heap aliases at compile-time. This is because *any* static indication that sharing may occur between expressions, requires the conservative assumption that sharing always occurs between the expressions, whether or not it actually occurs at run-time.

3.1.2 Linearization of Threads

Heap resolution detects access to shared structure when a thread t running a parallel version $\overline{p}$ of a procedure p accesses a popular node (a node with reference count greater than one). To decide whether the thread t may perform the shared heap access or must suspend until other threads complete, $\overline{p}$ consults a linearization of all active parallel threads [20]. A linearization is a total ordering of all active parallel threads that is a sequential schedule of the expressions under evaluation by these threads. If t is at the head of this linearization, the access takes place and evaluation resumes. If t is not the first thread in the linearization, t suspends until all threads ahead of t complete. Upon its completion, thread t removes itself from the linearization, thereby enabling later threads to access shared structure.

3.2 Programmer Declaration: *Acyclic*

In a language that allows side effects to the heap, a program can build cyclic structure. Cyclic structures make parallelization difficult since a thread with access to a node in a cyclic structure may "loop back" onto itself. Consider the two-element list `l=[a,b]` where `a` and `b` represent arbitrary heap structure. With cycles, it is possible that `a` reaches `l` (*e.g.*, `a = l`). A destructive procedure cannot be safely mapped over `l` in parallel since `a` reaches `b` through `l`. Furthermore, sharing between `a` and `b` cannot be detected dynamically using reference counts since the standard representation of the list `l=[a,b]`,



contains no popular nodes indicating sharing. If, however, the spine of `l` is known to be acyclic (neither `l` nor `l'` lie on a cycle in the heap) at compile-time, simple static analysis (§3.3) shows that `a` and `b` can reach shared structure only through a popular node, which can be detected at run-time. Note that declaring `l` as acyclic does not require `a` and `b` to be acyclic.

A programmer typically is aware of cyclic structure since precautions must be taken when iterating over it. Lists, tuples, trees and dags can easily be identified as acyclic by the programmer. To enable heap resolution, we introduce the *acyclic* datatype qualifier. An acyclic datatype is an ML datatype whose spine, *i.e.*, the heap nodes created by tuple and record constructors in the datatype, is guaranteed by the programmer to not lie on a cycle in the heap. This declaration implies that a spine element of an acyclic datatype does not reach itself. For example, the datatype

acyclic `datatype 'a pair = Pair of ('a * 'a)`

disallows a `pair` from reaching itself. Two `pair` elements can reach shared structure only if all paths from a paired element to the shared structure contain a popular node.

3.3 Analysis for Heap Resolution

The static analysis required for heap resolution consists of two parts: detection of candidate expressions for parallel evaluation and subsequent analysis of these expressions to determine static relationships among heap variables they use.

3.3.1 Expression Selection

Heap resolution selects a set of control-independent expressions, $\{e_1, \ldots, e_n\}$, whose parallel evaluation potentially conflicts due to heap writes (read and write effects). The compiler must know the set of variables through which an e_i accesses the heap. Therefore, procedures invoked by e_i may not have access to the heap through global heap variables or heap values in their closures (curried parameters). The following definition characterizes the procedures an e_i may invoke:

Definition 3 Procedure p is a *true function* if p does not reference free variables and p does not invoke procedures that reference free variables not bound in p.

A true function is a procedure that operates on state transmitted entirely through its parameters. For ex-

ample, the procedure:

```
fun f (a,b) =
  let fun g x = (x,a) in
      g b
  end
```

is a true function whereas procedure g is not since g uses the free variable b. The procedure qs (Figure 3) is a true function if its parameter cmp is a true function.[6] An $e_i \in \{e_1, \ldots, e_n\}$ may only invoke true functions. If e_i only invokes true functions, any state accessed and modified by e_i is completely described by e_i's free variables. Let $FV(e)$ denote the set of free variables in an expression e. The free variables $FV(e_i)$ denote the "inputs" to expression e_i. Subsequent analysis (§3.3.2) examines e_i's set of free heap variables,

$H_i = \{h_i \mid h_i \in FV(e_i)$ and h_i is a heap variable$\}$

and determines the structural relationship among the H_i sets.

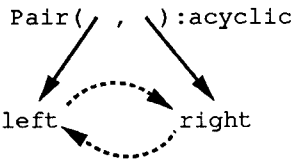
In the destructive quicksort of Figure 3, the expressions e_1=(qs left cmp) and e_2=(qs right cmp) are control-independent. $FV(e_1)$ = {qs, left, cmp} and $FV(e_2)$ = {qs, right, cmp}. The procedure qs is a true function if cmp is a true function. If cmp is a true function, then H_1 = {left} and H_2 = {right} represent the heap variables available to e_1 and e_2 respectively. The analysis of the following section deduces relationships between heap nodes dynamically bound to the heap variables in H_1 and H_2 and schedules e_1 and e_2 in parallel using heap resolution.

3.3.2 Local Structure Analysis

Heap resolution requires static detection of heap variables that reach shared structure only via paths certain to contain popular nodes. The analysis in this section statically verifies that the free heap variables H_i of expression e_i can access structure accessible to e_j (through its free heap variables H_j) only if they first encounter a popular node, for all i and j ($i \neq j$). If this property cannot be verified, the expressions $\{e_1, \ldots, e_n\}$ are rejected as candidates for heap resolution and must evaluate sequentially.

ML decomposes dynamic data by matching it against patterns. A data pattern recursively consists of variables, constants, constructors and patterns [15]. When a pattern matches a piece of dynamic data, variables in the pattern are bound to the piece of the data they represent. If the pattern contains acyclic datatype constructors, the dynamic relationship of heap variables within the pattern can often be deduced statically. The relationship of variables within patterns and relations among multiple patterns is captured in a *structure graph*. A structure graph for an expression e is a directed graph $G = (V, E)$ that describes the structure of patterns lexically visible to an expression e. G's vertices V consist of variables and patterns visible to e. Heap resolution requires free heap variables of e to correspond to variables in G. Edges E in G are of two types: pointer edges and path edges. A pointer edge represents a statically known pointer between a pair of vertices. Path edges represent possible (but statically unknown) paths between a pair of vertices.

In the procedure qs (Figure 3), for example, the structure graph



corresponds to the pattern Pair(left,right) in the expression: val Pair(left,right) = split a x'. The graph reveals that any heap node bound to the variable left (right) can reach the heap node bound to right (left) only if the node bound to right (left) has a reference count greater than one (⟶ is a pointer edge and ⇢ a path edge). All paths from left (right) to right (left) must contain a popular node. Furthermore, if left and right reach shared structure, a path to this structure must contain a popular node. Therefore, shared structure accessible by left and right can be detected dynamically using reference counts and the expressions (qs left cmp) and (qs right cmp) are compiled to evaluate in parallel using heap resolution.[7] Note that if the pair datatype were not acyclic, the structure graph would contain path edges from left to the pattern and from right to the pattern. These additional edges would allow left (right) to reach right (left) without accessing a popular node (through the vertex representing the pattern) and dynamic detection of sharing with reference counts would not work.

This static analysis first annotates expressions in a procedure p with the structure graph that lexically reaches them. The expressions $e_1, \ldots, e_n$ must be annotated with the same structure graph G for parallel evaluation with heap resolution. For each pair of free-heap-variable sets, H_i and H_j ($i \neq j$), the analysis examines G to verify that for all $h_i \in H_i$ and $h_j \in H_j$,

[6] A run-time λ-tag (§2) that identifies higher-order parameters as true functions can be used to determine this property.

[7] A run-time optimization to reduce false sharing is applicable here. If left (or right) has a reference count equal to one, this count can be lowered to zero before evaluation of (qs left cmp) (or (qs right cmp)) and incremented to restore the true count after evaluation of the expression completes. This is valid since a single pointer to left or right must be from Pair(left,right). The destructive quicksort of Figure 3 requires this optimization to remove false sharing created by the return value of split.

h_i can reach h_j (and h_j can reach h_i) only via a path that contains a popular node in G. If a path from h_i to h_j (or from h_j to h_i) in G does not contain a popular vertex, heap resolution cannot be used since potential sharing cannot be dynamically detected using reference counts.

This analysis is simple, yet able to schedule parallel evaluation of a large class of expressions. It is particularly well suited for parallel structure traversals. It is, however, unable to detect relationships between unrelated patterns such as those representing multiple parameters to a procedure. It is also unclear how effective this analysis is when applied to many control-independent expressions involving many free heap variables. We are developing a more sophisticated static/dynamic analysis that addresses these issues.

3.4 Markers

The following is a case where heap resolution is not applicable: statically, it may be known that heap node h cannot reach heap node h', but that h' can reach h. Reference counts do not work in this situation since there is no guarantee of a popular node on a path from h' to h. A *marker* can be placed on h to indicate that h (and nodes accessible from h) are in use by a thread. A heap node's marker is dynamically examined with its reference count, as explained above. A marker identifies the thread t that placed it, thereby granting t access to the node. Markers allow the expression

```
let val p as Pair(a,_) = x in
    f a; g p
end
```

to evaluate `(f a)` and `(g p)` in parallel. After `a` is marked, `(g p)` may evaluate. If `g` does not access `a`, or `f` removes or relocates `a`'s marker, parallelism ensues.

4 Implementation and Results

We added λ-tagging and heap resolution prototypes to the SML/NJ compiler [1]. We also built a parallel ML system from SML/NJ, *sml2c*, and SML Threads. *sml2c* [19] is a code generator for SML/NJ that produces C code. SML Threads [4] provides thread creation and synchronization primitives. *sml2c* combined with SML Threads allowed us to execute ML programs on a 386-based 20-processor Sequent Symmetry shared-memory multiprocessor (for which SML/NJ does not directly generate native code). In order to retain portability across architectures supported by SML/NJ, the only modification made to the *sml2c* code generator is support for reference counts.

The parallel threads allocate storage from distinct sections of the heap. A single processor performs stop-and-copy garbage collection after suspending all other active processors. All reported times include the time required for garbage collection.

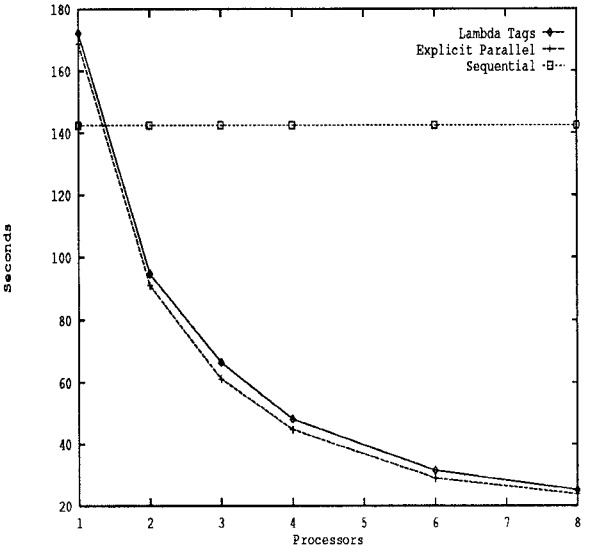


Figure 5: Timings of symbolic matrix multiply of two 100x100 integer matrices using λ-tags versus explicit fork/join parallelism. The multiply is parameterized by higher-order procedures that provide the addition and multiplication operations on matrix elements.

4.1 λ-tagging Results

To implement λ-tags, a single integer tag was added to the run-time procedure closures generated by SML/NJ. SML routines are used to inspect and modify a procedure's λ-tag. These routines correspond directly to the SAFE predicate and the tag-manipulation primitives SET, GET, and COMBINE (§1 and §2). Retrieving a pointer to a tag requires 39.7 μsec. Checking if a procedure's tag indicates no side effects requires 56.5 μsec, and modifying the tag requires 47.9 μsec. These times represent hundreds of machine instructions, whereas a direct implementation of these operations would require very few instructions and would further reduce the already small run-time overhead incurred by λ-tagging.

Figure 5 gives times for symbolically multiplying two list-based 100x100 integer matrices using λ-tags. Two higher-order procedures to the matrix multiply routine supply the operators for adding and multiplying matrix elements. If these operators are side-effect free, inner loops of the routine can evaluate in parallel (`map` is invoked repeatedly). In the test, neither higher-order operator had side effects. The program was manually

restructured to check λ-tags and re-assign λ-tags upon closure formation using the tag-manipulation primitives.

Figure 5 also compares our implicit parallel version of matrix multiply (through λ-tags) against an explicitly parallel version of the same routine. λ-tag times approach those of the explicitly parallel version. In this program, λ-tag overhead relative to the explicit parallel version ranged from 2% (one processor) to 9% (4 processors). This overhead is easily offset by the implicit parallelism obtained by our dynamic technique.

4.2 Heap Resolution Results

An implementation of heap resolution requires reference counting of pointers within the heap, a linearization of active parallel threads, and a primitive to inspect a heap node's reference count and suspend evaluation if necessary. A reference-count field was added to every dynamic SML/NJ record. The *sml2c* C backend was modified to increment count fields upon record construction and to decrement fields when pointers are re-assigned. Markers (§3.4) can be implemented similarly. The linearization of threads required to maintain the sequential semantics was implemented in ML as a doubly-linked list of thread descriptors. A primitive (written in C) was added to SML/NJ to create a new descriptor at a given location in the linearization. This operation is time critical since it occurs every time a thread is created. Insertion into the linearization may occur in parallel. Inlined SML procedures are used to inspect reference counts. If a thread t must suspend, t's continuation (available through SML/NJ's non-standard `callcc`) is stored with t's descriptor in the linearization. The continuation is invoked when t moves to the head of the linearization (after all prior threads have completed).

The destructive quicksort given in Figure 3 was restructured by hand to perform the recursive calls in parallel and to check reference counts. Figure 6a gives timings for sorting a list of 10000 random integers. The sequential version performed no reference counting. The overhead due to heap resolution in this program is 12% of the explicitly parallel time for 6 processors, but heap resolution provides better performance than sequential evaluation with only two processors.

The timings of a program to topologically sort a forest of trees are given in Figure 6b. The program sorted 25 balanced trees of depth 13. The trees did not share structure. Programmer or compiler parallelization of this program is difficult since sharing among the trees is unknown. The graph therefore lacks a curve with explicit parallel times. As reflected by the graph, this program is very pointer intensive, and heap resolution incurs significant overhead. Even so, heap resolution improves on the sequential performance when more than 4 processors are used. This program was also restructured manually.

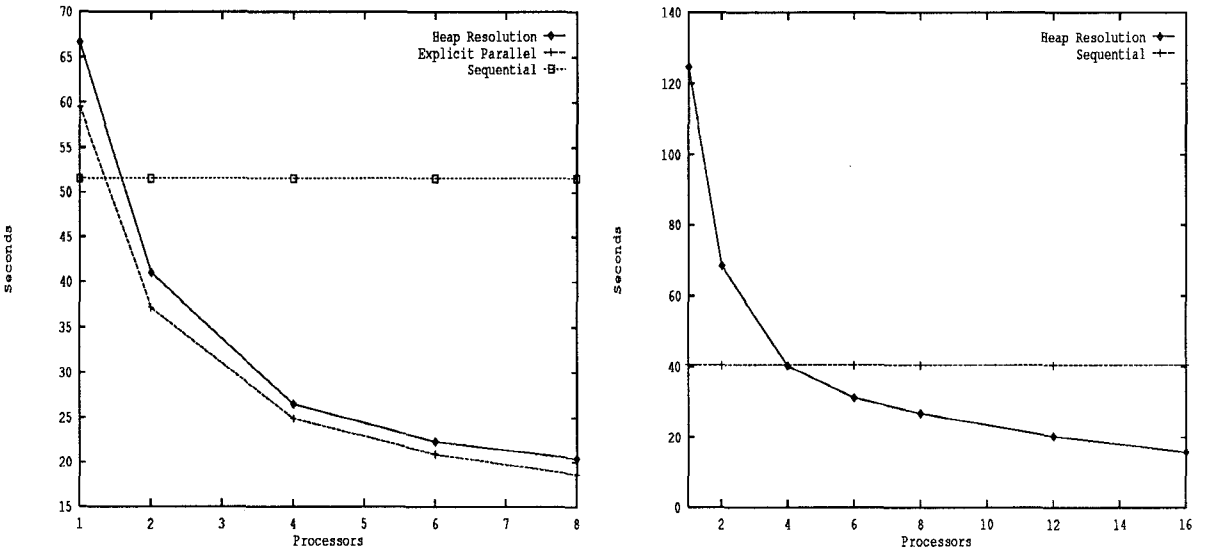
To measure the effect of sharing, the topological sort was applied to a forest of trees where each tree shared a leaf node common to all trees. Heap resolution required 37.2 seconds to perform the sort with 8 processors. This is still an improvement over the sequential sort (40.5 seconds). With sharing in the middle of the tree, the time required for the parallel sort increased to 65.0 seconds since most of the computation was performed sequentially with reference counts checked. This indicates that heap resolution in the presence of sharing is viable only if run-time overheads can be further reduced.

The overhead of heap resolution can be substantially reduced by allowing the lead thread in the linearization to read/write the heap with the conventional sequential code. Only threads not at the head of the linearization must respect reference counts. This allows heap resolution to operate one thread at original speed and only slows the progress of additional parallel threads that check reference counts. To do this effectively, an implementation must be able to switch between the conventional version of a procedure and the version that checks reference counts. This optimization has not yet been incorporated into our implementation. Additionally, examination of reference counts must be moved to the backend of the compiler, instead of being performed by a procedure call.

It is interesting to note that overhead due to maintaining λ-tags, a thread linearization, reference counts, and markers is "parallel" and the detrimental impact of this overhead diminishes as the number of processors increases. Even so, we expect the run-time overhead of λ-tagging and heap resolution to decrease as we refine the implementation.

5 Related Work

ParaTran [20] dynamically parallelizes Scheme by modeling heap accesses as database transactions and is most similar to our work. Evaluation in *Paratran* proceeds optimistically. Upon detection of a conflict, the computation must be *rolled back* to a point where the linear access order is intact. Reversing large computations is expensive. By contrast, our heap resolution technique suspends a conflicting expression and immediately begins evaluation of another pending expression in parallel. Heap resolution always makes forward progress. The amount of dynamic information required by heap resolution is small (reference counts)



(a) (b)

Figure 6: Timings of heap resolution applied to destructive quicksort (a) and topological tree sort (b). In both programs, heap resolution provides speedup over the sequential versions as the number of processors increases. No dynamic structures are shared. Explicit parallel timings for the topological sort are not given since this program is difficult to explicitly parallelize due to unknown sharing.

in comparison to the complex time-stamps *Paratran* retains for heap objects.

The Parcel [6] and CURARE [10] systems restructure Scheme after performing expensive data-flow analysis for detecting heap aliases. Complex heap dependences often force these static techniques to make safe, sequential assumptions. Higher-order functions are not rigorously traced by these techniques. However, static analysis of side effects in higher-order languages is addressed by Neirynck [16]. This conservative analysis is unable to effectively trace higher-order procedures propagated through data structures. The analysis required for λ-tagging is less expensive and potentially detects more safe higher-order applications at run-time than these techniques statically uncover. Recent work [3, 7, 8] addresses static analysis of heap structures, pointers, and recursive data structures. Work by Lu and Chen [11] uses a static analysis in conjunction with dynamic information to detect loop dependences between array references at run-time.

6 Conclusion

λ-tagging and heap resolution permit dynamic parallelization of expressive languages that admit higher-order procedures and allow side effects to the heap. These techniques reveal and exploit implicit parallel computations that are statically undetectable. Dynamic parallelization supports interactive development environments and separate compilation since static analysis is performed at the procedure, not program, level.

An implementation of λ-tagging and heap resolution in the SML/NJ optimizing compiler indicates that the costs inherent in run-time techniques—dynamic maintenance, propagation, and utilization of information—are more than offset by the dynamic discovery of parallel threads.

Acknowledgements

Thanks to Phil Pfeiffer and Todd Proebsting for insightful critiques of this work; to Andrew Appel, David Tarditi, and Greg Morrisett for assistance with SML/NJ, *sml2c*, and SML Threads respectively.

References

[1] A. W. Appel and D. B. MacQueen. A Standard ML compiler. *Functional Programming Languages and Computer Architecture*, 274:301–324, 1987.

[2] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 146–160, June 1989.

[3] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–310, June 1990.

[4] E. C. Cooper and J. G. Morrisett. Adding threads to Standard ML. Technical Report CMU-CS-90-186, School of Computer Science, Carnegie Mellon University, December 1990.

[5] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[6] W. L. Harrison. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, October 1989.

[7] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[8] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1989.

[9] P. Jouvelot and D. K. Gifford. Algebraic reconstruction of types and effects. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 303–310, January 1991.

[10] J. R. Larus. Compiling Lisp programs for parallel execution. *Lisp and Symbolic Computation*, 4:29–99, 1991.

[11] L. Lu and M. C. Chen. Parallelizing loops with indirect array references or pointers. In *Preliminary Proc. of the 4th Workshop on Languages and Compilers for Parallel Computing*, August 1991.

[12] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–57, January 1988.

[13] H. G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 382–401, January 1990.

[14] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[15] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[16] A. Neirynck. *Static Analysis and Side Effects in Higher-Order Languages.* PhD thesis, Cornell University, February 1988.

[17] J. Rees and W. Clinger (eds.). Revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.

[18] O. Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, CMU, May 1991.

[19] D. Tarditi, A. Acharya, and P. Lee. No assembly required: Compiling Standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, November 1990.

[20] P. Tinker and M. Katz. Parallel execution of sequential Scheme with Paratran. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 28–39, July 1988.