How Much Non-strictness do Lenient Programs Require?

Klaus E. Schauser
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
schauser@cs.ucsb.edu

Seth C. Goldstein
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720
sethg@cs.berkeley.edu

Abstract

Lenient languages, such as Id90, have been touted as among the best functional languages for massively parallel machines [AHN88]. Lenient evaluation combines non-strict semantics with eager evaluation [Tra91]. Non-strictness gives these languages more expressive power than strict semantics, while eager evaluation ensures the highest degree of parallelism. Unfortunately, non-strictness incurs a large overhead, as it requires dynamic scheduling and synchronization. As a result, many powerful program analysis techniques have been developed to statically determine when non-strictness is not required [CPJ85, Tra91, Sch94].

This paper studies a large set of lenient programs and quantifies the degree of non-strictness they require. We identify several forms of non-strictness, including functional, conditional, and data structure non-strictness. Surprisingly, most Id90 programs require neither functional nor conditional non-strictness. Many benchmark programs, however, make use of a limited form of data structure non-strictness. The paper refutes the myth that lenient programs require extensive non-strictness.

1 Introduction

Non-strict functional languages have been widely regarded as an attractive basis for parallel computing. Unlike sequential languages, they expose all the parallelism in a program. In side-effect–free functional languages, the arguments to a function call can be evaluated in parallel. Further, non-strict execution can substantially enhance parallelism, since functions can start executing before all of their arguments have been provided. In a language with *non-strict* semantics it is possible to define functions which return a result even if one of the arguments diverges. Among other things, this makes it possible to create recursively defined and cyclic data structures, since portions of the result of a function call can be used as arguments to that call. Non-strictness increases the expressiveness of the language but requires a more flexible strategy than strict evaluation, which evaluates all arguments before calling a function.

Non-strictness is usually combined with lazy evaluation, which delays evaluation of an expression until it is known to contribute to the result, thus decreasing parallelism. However, non-strictness can also be combined with eager evaluation. This combination, called *lenient* evaluation [Tra91], exhibits more parallelism than lazy evaluation while retaining much of the latter's expressive power.

Unfortunately, non-strictness comes at a high implementation cost, since it requires fine-grained dynamic scheduling and synchronization. Non-strict semantics can make it impossible to statically determine the order in which arguments are evaluated and operations execute. Expressions can be evaluated only after all of the arguments they depend on are available. This dynamic scheduling is expensive on commodity microprocessors, which efficiently support only a single thread of control and incur a high cost for context switching.

Much research has been devoted to analysis techniques that determine when the full generality of non-strictness is not required. These techniques include strictness analysis [Pey87], backwards analysis [Hug88b], path analysis [BH87], and partitioning [Tra91, SCvE91, HDGS91]. For lenient languages the compilation approach is to schedule instructions statically into sequential threads and have dynamic scheduling only between threads. The task of identifying portions of the program that can be scheduled statically and ordered into threads is called partitioning [Tra91]. Two operations can be placed into the same thread only if the compiler can statically determine the order in which they execute.

In [Sch94] we presented a new thread partitioning algorithm, separation constraint partitioning, which improves substantially on previous work [Tra91, HDGS91, SCvE91]. To evaluate our partitioning algorithm, we compared benchmark programs compiled using our algorithm and using strict partitioning. Strict partitioning ignores possible non-strictness and compiles functions and conditionals strictly, thus representing the best possible partitioning. Obviously, strict partitioning produces an incorrect partitioning (i.e., leading to a deadlock) for programs which require non-strictness. Surprisingly, almost all of our benchmark programs still ran correctly, an indication that non-strictness is rarely used in lenient programs.

This observation led us to pursue a more detailed study of how much non-strictness lenient programs use, and to focus on a larger set of programs. In this paper, we evaluate the strictness properties of a large collection of programs written in Id90, a non-strict functional language that uses lenient evaluation. We define three catagories of non-strictness: functional, conditional, and data-structure non-strictness. We find that functional and conditional non-strictness are rarely used in lenient programs. On the other hand, data structure non-strictness is used extensively. We

To appear in 7th Annual ACM SIGPLAN/SIGARCH Conference on Functional Programming Languages and Computer Architecture (FPCA'95).

further catagorize data-structure non-strictness into circular, recursive, and dynamically scheduled non-strictness, and we see that the expensive dynamically scheduled non-strictness is seldom required.

In this study our focus is on whether non-strictness is required (and if it is, what kind is required) for executing lenient programs correctly. Therefore, the study is not affected by whether the programs are executed in parallel or sequentially. We chose to study the sequential execution behavior. However, evaluating functions, even strict functions, non-strictly can increase parallelism. We find that non-strictness is rarely used, suggesting that future work on lenient languages should focus less on techniques for determining when non-strictness is not used, and more on when strict functions and conditionals should be evaluated non-strictly.

The remainder of the paper is structured as follows. Section 2 summarizes the different evaluation strategies for functional languages (strict, lenient, and lazy) and previous arguments for lenient evaluation as the most suitable way to write programs for massively parallel machines. Section 3 identifies and illustrates three kinds of non-strictness: conditional, functional, and data structure non-strictness. Section 4 describes the methodology and the benchmark programs which were used for this study, and presents the results which show the type of non-strictness each program requires. Section 5 discusses related work and Section 6 suggests directions for future work.

2 Lenient Evaluation

This section summarizes different evaluation strategies for functional languages: strict, lenient, and lazy. The three evaluation strategies differ in their expressiveness, parallelism, and efficiency. Many arguments have been made in favor of lenient evaluation, as embodied in the language Id90,[1] as the best choice for massively parallel machines [AHN88, Nik90]. Being non-strict, lenient evaluation retains much of the expressive power of lazy evaluation at a much lower overhead, while exhibiting more parallelism than both strict and lazy evaluation [Tra91].

2.1 The Lenient Language Id90

Id90 is a functional language augmented with synchronizing data structures. It consists of three layers, as shown in Figure 1. The functional core has all of the properties of purely functional languages, including referential transparency and determinacy. Id90 provides array comprehensions for efficiently expressing scientific computation requiring arrays or other large data structures. These are purely functional constructs and define an array by defining each element. With array comprehensions, elements of the array can be defined in terms of other elements. One limitation is that in order to be purely functional, the elements of the array have to be defined completely within the array comprehension. This ensures that referential transparency is preserved. Because this is not sufficient for certain problems [ANP87], Id90 also provides two non-functional data structures, I-structures [ANP89] and M-structures [BNA91].

I-structures are write-once data structures which separate the creation of the structure from the definition of its elements. Like functional arrays, I-structures provide efficient indexed access to the individual elements. In addition, I-structures provide element-by-element synchronization between the producer and consumer. I-structures are not purely functional, since they lose referential transparency. Any function obtaining a pointer to an I-structure may store into it. On the other hand, determinacy is preserved because each location can be stored into at most once. Non-deterministic computation can be expressed with M-structures, mutable data structures that provide support for atomic updates.

[1] Parallel Haskell (pH), an ongoing development which integrates many concepts of Id90 into Haskell, may also be based on lenient evaluation.

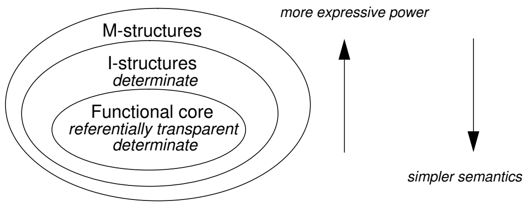

Figure 1: *The three layers of Id90 (from [Nik93]).*

2.2 Evaluation Strategies

The two most widely used evaluation strategies for functional languages are strict and lazy evaluation. We compare them to lenient evaluation, the strategy used in Id90. The three strategies differ in how they handle the evaluation of type constructors and function calls. As presented in [Tra91], to evaluate a function call $f\, e_1 \ldots e_n$,

Strict evaluation first evaluates all of the argument expressions e_1 to e_n, and then evaluates the function body, passing in the evaluated arguments;

Lenient evaluation starts the evaluation of the function body f in parallel with the evaluation of all the argument expressions e_1 to e_n, evaluating each only as far as the data dependencies permit;

Lazy evaluation starts the evaluation of the function body, passing the arguments in an unevaluated form.

The evaluation rules for constants, arithmetic primitives, and conditionals are the same for the three strategies. The evaluation of a constant yields that constant. Arithmetic operators are strict in their arguments, so they evaluate the arguments before performing the operation. The evaluation of a conditional *if* e_1 *then* e_2 *else* e_3 first evaluates the predicate e_1; depending on the result, either e_2 or e_3 is then evaluated. The three evaluation strategies treat the conditional conservatively: they do not start evaluating e_2 and e_3 speculatively before knowing the value of the predicate.

We see that under strict evaluation it is impossible to call a function with only some of the arguments. Thus one cannot pass the result (or part of it) back into the function. On the other hand, this is possible under both lenient and lazy evaluation, both of which implement non-strict semantics. Lazy evaluation ensures that only expressions which contribute to the final answer are evaluated; the evaluation is demand-driven. In contrast, strict and lenient evaluation evaluate all expressions (with the exception of those inside the arms of conditionals); they use eager evaluation, which is data-driven in the sense that an expression can be evaluated as soon as its data inputs are available.

As Table 1 shows, the three evaluation strategies differ substantially in their expressiveness, the amount of parallelism they expose, the implementation overhead, and the amount of speculative computation. These differences are discussed in the following subsections.

Expressiveness	strict	$<$	lenient	$<$	lazy
Parallelism	lazy	$\leq$	strict	$\leq$	lenient
Overhead	strict	$\leq$	lenient	$\leq$	lazy
Speculative comp.	lazy	$\leq$	lenient	$=$	strict

Table 1: *Comparison of the three evaluation strategies.*

2.2.1 Expressiveness

As shown in Figure 2, the expressiveness of the evaluation strategies forms a three-level hierarchy. Lazy evaluation results in more expressiveness than lenient evaluation, which in turn has more expressiveness than strict evaluation. By "more expressiveness," we informally mean that the translation of a program which uses non-strictness into a strict equivalent may require a global reorganization of the entire program; a more formal definition of expressiveness is given in [Fel91]. Non-strictness, as present in lenient and lazy evaluation, significantly increases expressiveness: The programmer can create circular data structures and define data structures recursively in terms of their own elements. In Section 3 we present several of such examples. Under the purely functional setting, non-strictness results in more efficient programs—in both space and time requirements [Hug88a, Bir84, Joh87]. In addition, lazy evaluation provides the control structure to handle infinite data structures, as long as only a finite part is accessed. Using explicit delays, programmers can obtain the same benefit under lenient evaluation [Hel89].

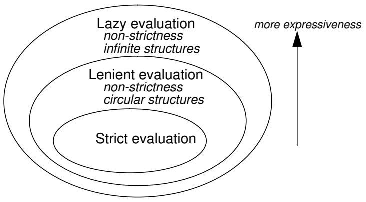

Figure 2: *Expressiveness of the three evaluation strategies.*

2.2.2 Parallelism

The evaluation strategy can have a strong impact on the amount of parallelism [HA87]. Lenient evaluation results in the largest amount of parallelism, while lazy evaluation shows the least amount of parallelism [Tre94]. Lazy evaluation needs to show that an argument is actually required for the final answer before starting its evaluation, while strict and lenient evaluation can always evaluate independent arguments in parallel. In addition, lenient evaluation can exploit producer-consumer parallelism. Consider the function `flat` which produces a list of the leaves of a binary tree using accumulation lists [Nik91].

```
def flat tree acc =
 if (leaf tree) then (cons tree acc)
 else flat (left tree) (flat (right tree) acc);
```

This example does not require non-strictness; therefore it can be executed with any of the three evaluation strategies. However, under strict evaluation this code exhibits little parallelism. The right subtree is completely flattened before starting with the left subtree. And this holds for every level, so the tree is flattened sequentially.

Under lenient execution the flattening of the two subtrees can be pipelined. Flattening of the right subtree can be started simultaneously with the flattening of the left subtree. The resulting list for the right subtree is fed into the function working on the left subtree. In effect, the entire list is constructed in parallel. This example shows that overlapping the consumer and producer can yield more than just a constant factor increase in parallelism [Nik91].[2]

Lazy evaluation flattens the subtrees in the opposite order from strict evaluation. Lazy evaluation first flattens the left subtree, and then starts working on the right subtree.[3] Therefore lazy evaluation also does not exhibit any parallelism.

2.2.3 Evaluation Overhead

As should already be clear, the evaluation strategies differ substantially in their overhead. The biggest differences are in the costs for argument passing and dynamic scheduling.

Strict evaluation has the lowest overhead. All arguments can be evaluated before calling a function, and therefore can be passed by value. Furthermore, it is possible to statically schedule all of the computation inside a function and produce efficient sequential code. The compilation of strict functional languages is therefore similar to well understood implementations of sequential languages.[4] Lazy evaluation has a high overhead because it requires arguments to be passed in unevaluated form, unless it can be shown that they contribute to the final result. The overhead of lenient evaluation falls between that of lazy and strict evaluation [Tra91].

2.2.4 Speculative Computation

Lazy evaluation is the only strategy which completely avoids speculative computation; it evaluates only expressions which contribute to the final result. The other two schemes do computation eagerly, thereby putting the burden on the programmer to avoid infinite or speculative computation.

2.2.5 Summary

We have seen that both lenient and lazy evaluation realize non-strictness, i.e., they may start the evaluation of a function body before evaluating the arguments. When non-strictness is combined with eager evaluation, as in lenient evaluation, parallelism is increased substantially, because all arguments can be evaluated in parallel with the function body. This is why many researchers have claimed that lenient evaluation is best for parallel implementation of functional languages.

[2] Simulations for the dataflow architecture TTDA show that with lenient evaluation the critical path on a full binary tree of depth 10 consists of 250 time steps with a maximum parallelism of 1776 and an average parallelism of 266 instructions (assuming the resources are available). If executed strictly, the critical path would grow to 26,650 time steps, with a maximum parallelism of 4 instructions.

[3] Of course, lazy evaluation would only produce as much of the flattened list as is actually required.

[4] Additional issues arise due to the use of higher-order functions, partial applications, parallelism, and single assignment limitations.

3 Forms of Non-strictness

In this section we identify different forms of non-strictness and present examples to illustrate them. The present work focuses on programs which execute under lenient evaluation and does not consider programs requiring lazy evaluation, i.e., those that manipulate potentially infinite data structures. Thus we limit our discussion of the various forms of non-strictness to lenient evaluation.

3.1 Functional Non-strictness

Functional non-strictness arises from feedback dependencies from the results of a function invocation to its arguments. To illustrate this form of non-strictness and the need for dynamic scheduling we use the following simple, but somewhat contrived, computation [SCG95].[5]

```
def two x y = (x*x, y+y);
def g z = { a,b = (two z a) in b};
def h z = { a,b = (two b z) in a};
```

In this example, the function *two* takes two arguments, x and y, and returns two results, $x*x$ and $y+y$. Inside the function *two* there is no dependence between the multiplication and addition; thus code to evaluate the two halves of the pair can be put in either order when compiling the function under traditional strict evaluation. This is not true under non-strict evaluation. In our example, the function *two* is used in two different contexts which require non-strictness. In the function g the argument z is given as the first argument to the function *two*, while the second argument to *two* is taken from its first result. This requires that *two* first compute $z*z$, return this result, and then compute $(z*z)+(z*z)=2z^2$. We see that in this case the multiplication is executed before the addition. In the function h just the opposite occurs: The second result of the function *two* is fed back in as the first argument. Here $z+z$ is computed first and then $(z+z)*(z+z)=4z^2$. Now the addition is executed before the multiplication. Thus the multiplication and the addition have to be scheduled independently. Notice that the scheduling is independent of the data values for the arguments; it depends only on the context in which the function is used and how results are fed back in as arguments.

3.2 Conditional Non-strictness

Non-strictness and the requirement for dynamic scheduling not only occur across function calls, but can also appear within conditionals. The following example, taken from [Tra91], illustrates this.

```
def kt p z = { a,b,c = if p then (y,z,x) else (z,x,y);
               x = a+a;
               y = b*b
              in c};
def g z = kt true z;
def h z = kt false z;
```

In this example a single conditional steers the evaluation of three variables, a, b, and c. If the predicate is true then b gets the value of z, y the value $z*z$, and a the same value as y; x and the result, c, become $z*z+z*z=2z^2$. In this case the multiplication is executed before the addition. If the predicate is false the variables are evaluated in a different order. First the variable a is bound to the argument z, then x and b evaluate to $z+z$, and finally y and the result, c, evaluate to $(z+z)*(z+z)=4z^2$. Now the addition is performed before the multiplication. Again, we see that both the addition and the multiplication have to be scheduled dynamically. Though the operations appear outside of the scope of the conditional, the conditional affects the order in which the values a, b, and c are available.

It may seem that in this example, unlike the previous one, the scheduling is at least data-dependent, since it is influenced by the conditional and therefore depends on the value of the predicate. While this observation is correct, we can obtain precisely the same behavior without conditionals as shown in the next example.

```
def f1 x y z = (y,z,x);
def f2 x y z = (z,x,y);
def kt2 func z = { a,b,c = (func x y z);
                   x = a+a;
                   y = b*b
                  in c};
def g z = kt2 f1 z;
def h z = kt2 f2 z;
```

Here the conditional is replaced with a call to a function taking three arguments and returning three results. The argument *func* determines which function is called; it is $f1$ in the case of g and $f2$ in the case of h. These two functions, $f1$ and $f2$, do not perform any computation; they just shuffle the results around and thereby affect the order in which the addition and multiplication in the caller get executed. This example illustrates that in addition to the caller affecting the order in which operations get executed in the callee, the callee can also affect the order in the caller. In general, it is the whole context—i.e., the whole call tree—in which a function appears which determines the order. This example also shows the duality between conditionals and function calls made through function variables. A conditional can be viewed as a call site, where one of two functions is called depending on the predicate [AA89]. These two functions contain the code of the *then* side and *else* side of the conditional, respectively.

3.3 Data Structure Non-strictness

In non-strict languages, data structure constructors exhibit the same form of non-strictness as function calls, i.e., the result may be required before all the elements are defined. This gives the programmer the ability to define circular data structures or recursively define some elements in terms of other elements. For example, the recursive binding `a = (cons 1 a)` denotes a simple cyclic list, and `a = (cons 2 (hd a))` denotes an 2-tuple containing the same element. This power of non-strictness also extends to list comprehensions, array comprehensions, and I- and M-structures [AHN88, ANP89, BNA91].

We can define the following hierarchy of non-strictness in data structures.

Functionally strict: All elements must be evaluated before the data structure is created.

Circular: A pointer to a data structure may be stored into one of its elements.

[5] In all of the examples we present, the different definitions of the function g always perform the same computation $(z*z)+(z*z)=2z^2$. Similarly, the different definitions of the function h always compute $(z+z)*(z+z)=4z^2$. In all cases a single addition and single multiplication are executed; in the functions g the multiplication occurs before the addition, while in the functions h they execute in the reverse order.

Recursive: An element of a data structure may be defined in terms of other elements. For this case we can distinguish two subcases which describe the schedule used to fill in the elements.

Static: A static schedule can be found for the program which fills in the elements in the right order.

Dynamic: A dynamic schedule is required. (This implies that presence bits are needed, as fetches from elements may defer.)

The two small examples above using `cons` can be classified as circular and recursive, respectively. Other frequently cited examples using recursively defined data structures are wavefront [ANP89] and the following function computing an array of the first n Fibonacci numbers.

```
def fib_array n = { array (0,n) of
                  | [0] = 1
                  | [1] = 1
                  | [i] = A[i-1]+A[i-2] || i <- 2 to n};
```

Both examples can be scheduled statically, i.e., a static scheduling of the program can be found which obeys the data dependencies among the array elements (left-to-right in the case of fib_array and left-to-right, top-to-bottom in the case of wavefront). With such a schedule it is possible to implement the arrays without any presence bits (just like plain arrays in standard imperative languages). This may not always be the case, as the following example illustrates.

```
def dyn A k l m n = { A[k] = A[m] * A[m];
                      A[l] = A[n] + A[n]};
def g z = { A = I_array (1,3);
            A[1] = z;
            _ = (dyn A 2 3 1 2);
          in A[3] };
def h z = { A = I_array (1,3);
            A[1] = z;
            _ = (dyn A 3 2 2 1);
          in A[3] };
```

This example shows that dynamic scheduling may be required for non-strict I-structures. The function *dyn* takes five arguments—an I-structure A and four indices k, l, m, and n. It fetches an element from $A[m]$, multiplies this with itself, and stores the result into location $A[k]$. The function also fetches from location $A[n]$, adds this element to itself, and stores it into location $A[l]$. Figure 3 shows the dependencies for the body of function *dyn*, and what happens for the two cases where $k = n$ and $l = m$.

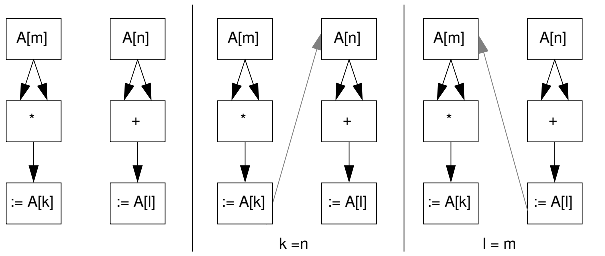

Figure 3: *The dependencies for function* dyn *and two possible scenarios,* $k = n$ *and* $l = m$.

If $k = n$, as is the case when *dyn* is called from the function g, then the store into location $A[k]$ defines the value which is fetched from $A[n]$. So there is a dependence from the store to the fetch (as indicated by the dashed line), and the multiplication has to be executed before the addition. Note that this dependence is not directly present in the function, but is established through the synchronizing I-structure. Should the fetch occur before the store, it would get deferred until the store happens. In the case where $l = m$, the operations would execute in the reverse order. Obviously, the conditions $k = l$ and $l = m$ should not hold together, since deadlock would result.

In general, it is not known which function fills in an I-structure; the consumer of an I-structure element cannot name its producer. Therefore if a fetch defers, it is necessary to continue with the evaluation of any expression which does not depend on the fetch, even if the computation is in a different function activation. This example illustrates that dependencies have to unravel at run time and that they may travel not only through arguments, results, and internal call sites, but also through I-structure accesses.

4 Experimental Results

In this section we assess the degree of non-strictness required by a large set of Id90 programs. Most of these programs were implemented by other researchers.

4.1 Methodology

To determine how much functional or conditional non-strictness our programs require, we extend the compiler with options to treat function calls and conditionals in a strict fashion. Under this mode, a function is invoked only after all of its arguments have been evaluated, and results are returned only after they have been completely computed. Similarly, we can also compile conditionals so that they are evaluated only after the predicate and all inputs are available, and return only after all results have been produced. In the case where a program requires functional or conditional non-strictness, this compilation scheme leads either to a static dependence cycle (which the compiler detects) or to deadlock at run-time. In either case we know that the program requires non-strictness.

To check for data structure non-strictness we extend the run-time system to handle synchronizing data structures differently. These variations in the run-time system emulate the different degrees of data structure non-strictness. For example, to check whether elements are recursively defined, we make structures fully strict in all elements. A fetch from a structure is deferred until all structure elements have been filled in. The run-time system reports if the program deadlocks, which is an indication of recursive dependencies among elements of the same structure. Similarly, to check whether a program can execute using a static schedule we test whether any deferred fetches occur. If no fetches defer, this is an indication that the current schedule fills in the data elements in the right order. Obviously, this approach would determine only that a static schedule exists for the current input of the program under the current partitioning and execution order. To determine that this is also the case for any input, we study the program in detail to ensure that data dependencies are not related to problem size and other input parameters (for example, *wavefront*), or make sure that all possible data dependencies are generated (for example, *dyn*). As it turns out, most of the original programs execute with defers because they are inherently parallel and the compiler and run-time system just happen to schedule a producer and consumer in the wrong order. We solved this by adding explicit sequentialization statements (barriers) to the program to ensure the correct order of execution.

4.2 Compilation Framework

The programs were all compiled using the Id90 compiler with the Berkeley TAM back-end. The compiler uses a front-end developed at MIT [Tra86] which produces dataflow graphs. The back-end partitions these dataflow graphs into threads and then generates code for TAM, a Threaded Abstract Machine [CGSvE93]. The TAM code is then translated to the target machine. Our translation path uses C as a portable "intermediate form" and produces code for the CM-5 as well as for various standard sequential machines [Gol94]. For the experiments performed in this paper we limit ourselves to sequential execution on a SparcStation 10. As we have explained, the compiler has been extended with options to treat function calls and conditionals strictly, as well as with a run-time system to test for the different degrees of non-strictness in data structures.

4.3 Programs

We used the benchmark programs shown in Table 2. The programs vary substantially in size and application area as well as in behavior. They range in size from a few to several thousand lines of source code. The small examples discussed so far in this paper are included as a point of reference. The application areas represented by these programs include scientific computing, sorting and search problems, symbolic computing, NAS parallel benchmarks, and small kernels. Most of the programs fall into the catagory of scientific computing, the area specifically targeted by the implicitly parallel language Id90 [AE88]. Most of the programs contain many conditionals and function calls and exhibit fine-grained behavior (e.g., Quicksort), while programs such as the blocked matrix multiply are more medium-grained [SGS+93].

4.4 Non-strictness Requirements

Following the methodology described above we determined the degree of non-strictness each of the programs requires. The results of our measurements are summarized in Table 3. The first two columns indicate whether the program contains any functional or conditional non-strictness. The third column shows whether the program has any circular data structure definitions. The fourth column shows whether any data element is defined recursively in terms of other elements of the same data structure. Finally, the last column indicates whether the program requires dynamic scheduling. Often we were able to find a static scheduling of the program (possibly by inserting barriers) that would allow it to execute without deferred fetches.

Most of the Id programs still run correctly after they are compiled with strict functions and strict conditionals. This indicates that programmers rarely make use of the additional expressiveness provided by non-strictness at the functional or conditional level. On the other hand, many of the benchmark programs make use of a limited form of data structure non-strictness. The ability to define circular data structures or array elements in terms of other elements is certainly important.

For most of the programs we were able to find a schedule that results in no fetches deferring at run-time. This result is contrary to the current thought that lenient programs inherently require dynamic scheduling.

Program	Lines	Short Description and Reference
Symbolic computation		
Paraffins	185	Enumerate isomers of paraffins [AHN88]
Primes_filter	40	Generate primes by filtering [Hel89]
Primes_sieve	50	Primes using sieve of Eratosthenes [Tre94]
Compresspath	89	Recursive doubling algorithm [SPR90]
N-Queens	66	N-queens problem (using circular lists) [All93]
NQ_Solutions	66	Number of N-queens solutions [All93]
Permutations	57	Generate permutations of sets of elements (using circular structure) [All93]
Id Compiler	38919	Id in Id compiler
Id Interpreter	740	Top level Id interpreter
Quicksort	55	Quick sort on lists [CSS+91]
Arraysort	75	Selection sort on arrays [CSS+91]
Bitonicsort	142	Bitonic sort (M-structures) [SB93c]
Bubblesort	67	Bubble sort (M-structures) [SB93c]
Heapsort_IS	128	Heap sort (I-structures) [SB93c]
Heapsort_MS	123	Heap sort (M-structures) [SB93c]
Quicksort_IS	124	Quick sort (I-structures) [SB93c]
Quicksort_MS	115	Quick sort (M-structures) [SB93c]
Mergesort	103	Merge sort (I-structures) [SB93c]
Shellsort	90	Shell sort (M-structures) [SB93c]
Scientific computation		
Wavefront	40	Simple wavefront SOR [ANP89]
Pseudoknot	3323	Molecular Biology [HFA+94]
Gamteb	649	Monte Carlo neutron transport [BCS+89]
MCNP	2351	Monte Carlo photon transport [HL93]
Simple	1105	Hydrodynamics and heat conduction [AE88]
Speech	172	Speech processing [Sah91]
DTW	100	Dynamic time warp [Sah91]
MM	74	Matrix multiply [HCAA93]
MMT44	118	Blocked matrix multiply test [CGSvE93]
Eigen3	151	Eigen problem
Householder	304	Householder Eigen-Solver [SB93a]
Jacobi	215	Jacobi Eigen-Solver [BH93]
Jacobi_group	162	Jacobi Eigen-Solver (group rotations) [BH93]
FT_fu	370	NAS benchmark FT (functional) [SB93b]
FT_is	333	NAS benchmark FT (I-structures) [SB93b]
FT_ms	356	NAS benchmark FT (M-structures) [SB93b]
Small examples		
KT-cond	5	Traub's non-strict conditional [Tra91]
Two	5	Non-strict function with two results [SCG95]
Cube	6	Non-strict cube function [SCvE91]
Flat	17	Flatten leaves of tree [Nik91]
Fib	3	Doubly recursive Fibonacci
Fib_lazy	5	Infinite list of Fibonacci numbers
Fib_lenient	6	Finite list of Fibonacci numbers
Fib_strict	5	Fibonacci using linear recurrence
Fib_array	5	Fibonacci using functional arrays

Table 2: *Benchmark programs. The programs are available from http://www.cs.ucsb.edu/~schauser/id90.*

Program	Form of non-strictness Func.	Cond.	Circ.	Recu.	Dyn.
Symbolic computation					
Paraffins	No	No	No	Yes	No
Primes_filter	Yes	No	No	Yes	Yes
Primes_sieve	No	No	No	No	No
Compresspath	No	No	Yes	No	No
N-Queens	Yes	No	No	Yes	Yes
NQ_Solutions	Yes	No	No	Yes	Yes
Permutations	Yes	No	No	Yes	Yes
Id Compiler	Yes	Yes	Yes	Yes	Yes
Id Interpreter	No	No	No	Yes	No
Quicksort	No	No	No	No	No
Arraysort	No	No	No	No	No
Bitonicsort	No	No	No	Yes	No
Bubblesort	No	No	No	Yes	No
Heapsort_IS	No	No	No	Yes	No
Heapsort_MS	No	No	No	Yes	No
Quicksort_IS	No	No	No	Yes	No
Quicksort_MS	No	No	No	Yes	No
Mergesort	No	No	No	Yes	No
Shellsort	No	No	No	Yes	No
Scientific computation					
Wavefront	No	No	No	Yes	No
Pseudoknot	No	No	No	No	No
Gamteb	Yes	No	No	Yes	Yes
Gamteb(defsubst)	No	No	No	Yes	No
MCNP	No	No	No	Yes	No
Simple	No	No	No	Yes	No
Speech	No	No	No	Yes	No
DTW	No	No	No	Yes	No
MM	No	No	No	No	No
MMT44	No	No	No	No	No
Eigen3	No	No	No	No	No
Householder	No	No	No	Yes	No
Jacobi	No	No	No	No	No
Jacobi_group	No	No	No	No	No
FT_fu	No	No	No	Yes	No
FT_is	No	No	No	Yes	No
FT_ms	No	No	No	Yes	No
Small examples					
KT-cond	No	Yes	No	No	Yes
Two	Yes	No	No	No	Yes
Cube	Yes	No	No	No	Yes
Flat	No	No	No	No	No
Fib	No	No	No	No	No
Fib_lazy	Lazy	No	No	No	Yes
Fib_lenient	Yes	No	No	No	Yes
Fib_strict	No	No	No	No	No
Fib_array	No	No	No	Yes	No

Table 3: *Non-strictness requirements. The entries indicate whether the program requires functional, conditional, or data structure non-strictness. Data structure non-strictness is further divided into circular, recursive, and dynamic non-strictness.*

4.5 Discussion

We now present the results in more detail, focusing on the individual programs and the different forms of non-strictness.

4.5.1 Functional Non-strictness

Few programs require functional non-strictness, and most of those that do were written precisely to exhibit this behavior. The two exceptions are Gamteb and the Id Compiler, which we discuss in more detail below.

For the small examples, the non-strict functions *two* (presented in Section 3.1), the non-strict Cube, and the lenient version of Fibonacci use functional non-strictness. We have also included a lazy version of Fibonacci which generates the infinite list of Fibonacci numbers and then returns the nth number. Under our lenient evaluation, this program fails to return the answer because it runs out of heap space while generating the infinite list. There are also several medium-sized programs—Primes_filter, N_Queens, NQ_Solutions, and Permutations—which use functional non-strictness. They use it in a very similar way, which is most easily illustrated with the lenient Fibonacci function.

```
% list of Fibonacci numbers from i to n
def fib_list l i n =
    if i > n then nil else
      (((nth (i-1) l)+(nth (i-2) l)) :
       (fib_list l (i+1) n));
def fib n = { li = (1 : 1 : (fib_list li 2 n));
             in nth n li };
```

This function creates the list of Fibonacci numbers from 1 to n and then returns the nth number. The function that creates the list of Fibonacci numbers from i to n requires the Fibonacci numbers below i so that it can compute the subsequent numbers. This is achieved by providing the base list for the first two numbers and then feeding the result list `li` back in as an argument to `fib_list`. The programs Primes_filter, N_Queens, NQ_Solutions, and Permutations are based on the same principle.

The largest program requiring functional non-strictness is the Id Compiler. Non-strictness is used to create circular data structures, where one element of a data structure contains a reference to the data structure itself. One such pattern which occurs frequently is found in the code for creating variable structures that contain information pertinent to Id identifiers. This is done using a record which contains fields about the identifier. Some of the record fields are defined as functional properties (defined at creation time and not modified thereafter), some are I-structure properties (emtpy at creation time, to be filled in later at most once), and some are M-structure properties (mutuable, capable of repeated modification). As shown in the example code below, one of the functional fields, `vr_node`, includes a reference to the record itself, thus introducing non-strictness.

```
def make_var_struct sourcename place arity opcode =
 {vs = {record
         ...    % other entries
         vr_node = (func (vs:nil));
       };
  In vs
 };
```

One way of avoiding the mutual recursion would be to change the functional field `vr_node` into an I-structure field. When creating the variable structure record this field would initially be left empty. After the record is created, `func` could be invoked and the result used to fill in the field `vr_node`. Of course, this solution goes beyond the purely functional programming style, but the non-functional aspect is not visible outside of this function.

Another interesting program in this category is Gamteb, which requires functional non-strictness because it uses the library-defined accumulators to tally statistics. Gamteb allocates several of these accumulators using the function `make_accumulator`.

```
result, accum  =  make_accumulator (1,35) (+) done;
```

This library function takes three arguments: the bounds of the accumulator, the operation used for accumulation, and finally a trigger to indicate that the accumulation has finished. The function returns two results: the accumulator structure which is used to perform the accumulations (`accum`) and the array which contains the final result (`result`). The accumulation is done using M-structures, which perform the updates in place. It is important that the final result be returned only after all of the accumulations are done. Otherwise a consumer of the result may read incorrect values while the accumulation is still going on. Thus the function `make_accumulator` exhibits the classical form of non-strictness. It has to return the accumulation structure after receiving the bounds and the operation, and can return the final result only after receiving `done` when all accumulations have completed. Trying to pass in all three arguments together will result in deadlock.

Nevertheless, it is possible to modify the program so that it does not use functional non-strictness.[6] What is required is to separate the creation of the accumulator from the closing of it. Then we can call them by separate functions and have the programmer ensure that they execute in the correct order (e.g., using explicit barriers).

4.5.2 Conditional Non-strictness

Only two programs require conditional non-strictness. The first is the small function presented in Section 3.2, specifically developed by Traub to illustrate the concept of non-strictness going through conditionals. The second is the Id Compiler which uses conditional non-strictness to conditionally create circular data structures. As shown below, the example is very similar to the code presented in the previous subsection. But this time, depending on the arity argument one of two records is created.

```
def make_var_struct2 sourcename place arity opcode =
{vs = if (eq 0 arity) then
       {record
         ....            % entries for arity equal 0
         vr_node = (func (vs:nil));
       }
       else
       {record
         ....            % entries for arity not 0
         vr_node = (func (vs:nil));
       };
 In vs
};
```

It is straightforward to change this code to avoid using conditional non-strictness. All we need to do is fully define the record in each arm of the conditional. For example it could be changed into the following.

```
def make_var_struct2 sourcename place arity opcode =
{vs = if (eq 0 arity) then
        {vst = {record
                 ....  % entries for arity equal 0
                 vr_node = (func (vst:nil));
               }
             in vst}
       else
        {vse = {record
                 ....  % entries for arity not 0
                 vr_node = (func (vse:nil));
               }
             in vse};
  In vs
};
```

[6] In fact, the version of Gamteb that we tested did not exhibit functional non-strictness as the calls to allocate the accumulators were inlined into the caller using Id90's defsubst mechanism. Only when inlining was disabled did we encounter functional non-strictness.

Our results seem to be a strong indication that, except in the simplest cases, conditional non-strictness is too complicated for programmers to use. Just tracing the dependencies of the small example given in Section 3.2 already requires substantial effort. Unfortunately, conditional non-strictness has a serious impact on code-generation. The compiler generates an independent switch for every input to a conditional and an independent merge for every output. This results in a large run-time overhead. This can be avoided only if the compiler does the appropriate analysis to figure out that the conditional non-strictness does not occur.

4.5.3 Data Structure Non-strictness

Data-structure non-strictness seems to be the most important form of non-strictness. Many programs use it. Some create circular data structures, but most define array elements recursively in terms of other elements. The prototypical example is wavefront or the `fib_array` presented in Section 3.3. The ability to define data structures recursively is essential for efficiency. Without this feature, defining a new element would require subsequent allocation of new copies of the arrays. Obviously, imperative languages have provided the power of non-strict data structures all along. In imperative languages, the programmer can allocate a data structure and then fill it in any order; values can even be updated in place.

One feature imperative data structures do not have is presence bits to provide synchronization on an element-by-element basis. The programmer has to explicitly ensure that all dependencies are satisfied and that the structure is filled in the right order. In Id90 presence bits automatically ensure the right order. For almost all of the programs, we were able to find a static schedule such that fetches of data elements would never defer. Often we were required to enhance our benchmark programs with explicit sequentialization statements (we used barriers) to ensure the right execution order.

Non-strict data structures are critical and are required for efficient execution. The lenient Fibonacci example from Section 4.5.1 and the Fibonacci array from Section 3.3 illustrate this. Both construct a data structure with the first n Fibonacci numbers (either a list or an array), and then return the nth number. This is the idea underlying many dynamic programming problems [Tra91]. Thus one of the reasons we don't see more functional non-strictness may be that we have data structure non-strictness, which gives us a similar expressive power and often can express complicated dependencies more elegantly and efficiently.

5 Related work

As far as we know, this is the first study to focus on how much non-strictness lenient programs require.

Tremblay did a similar study for lazy evaluation [Tre94]. He collected a large set of benchmark programs and determined whether they actually require lazy evaluation or whether lenient evaluation suffices. Lazy evaluation is required only if the program manipulates infinite lists. His observation is that for most programs lenient evaluation is sufficient. He found that, contrary to what other authors claimed, most of the programs presented in [Bir84, All92, All93] do not require laziness. Tremblay also studied how lazy evaluation impacts parallelism (under the TTDA execution model).

Much research has been devoted to analysis techniques that determine when the full generality of non-strictness is not required. These techniques include strictness analysis [Pey87], backwards analysis [Hug88b], and path analysis [BH87]. For lenient languages this is approached by partitioning a program into sequential threads [Tra91]. Partitioning algorithms have been developed by [Tra91, SCvE91, HDGS91, SCG95].

Many researchers have studied the parallel execution of a variety of lenient programs [AHN88, Hel89, CSS+91, SB93c, ANP89, AE88, Sah91, HCAA93, CGSvE93, SB93a, BH93, SB93b, SCG95, Nik91, SGS+93]. These studies have been performed on various hardware and simulation platforms, including the TTDA dataflow machines, Monsoon architecture, and MINT simulator, as well as TAM implementations on the CM-5, Sequent, and J-Machine. All of the programs used in those studies are included in this paper.

6 Conclusions

This paper studies a large set of Id90 benchmark programs and determines the degree of non-strictness they require. Contrary to current thought, most of the programs require neither functional nor conditional non-strictness. The only large program which requires these forms of non-strictness is the Id compiler, which uses them to define circular data structures. We believe that both functional and conditional non-strictness are seldom used because they are difficult to grapple with mentally: The seemingly cyclic dependencies from results back to arguments automatically unravel at run-time and are difficult to trace. Another reason for the lack of functional non-strictness may be that programmers tend to follow the imperative programming style they are used to. For example, many of the larger programs we studied fall into the area of scientific computation and may have been based on FORTRAN equivalents (e.g., Gamteb and Simple).

We think, however, that the main reason for the lack of functional non-strictness lies elsewhere. Even a limited use of non-strict data structures, viz. the ability to define data structure elements recursively, provides essentially the same expressiveness as, and often can encode these recursive dependencies more efficiently than, purely functional non-strictness using lists. Thus many of the programs make use of data structure non-strictness. An unexpected result is that for many of the programs we were able to determine a schedule such that none of the fetches to data structures deferred. This seems to indicate that at least a sequential implementation of lenient languages might be able to eliminate much of the overhead in managing the presence bits of synchronizing data structures.

One form of non-strictness that lenient evaluation does not provide is support for streams and infinite data structures. While we think that this may be very useful, we would caution against making a language lazy just for this reason. First, by using explicit delays or control constructs, programmers can obtain the same benefit without lazy evaluation [Hel89]. Second, Tremblay has observed that many supposedly lazy programs require only lenient evaluation [Tre94].

Since funtional and conditional non-strictness are rarely used and often unnecessary, we suggest eliminating them from the language. We believe that data structure non-strictness is sufficient. Researchers should devote more effort to developing techniques which determine when strict functions and conditionals should be evaluated in a non-strict fashion to beneficially increase the parallelism. As the example `flat` from Section 2.2.2 clearly shows, lenient evaluation can expose large amounts of consumer-producer parallelism which may be completely lost under strict evaluation. While idealized parallelism profiles derived from dataflow simulations such as the TTDA are capable of exposing the inherent parallelism in a program under lenient evaluation, more realistic methods have to be used to gain information about parallelism which reflects more accurately the execution on real parallel machines.

We conclude that while it is desirable to exploit the increased parallelism under lenient evaluation, is not necessary to make the language non-strict.

Acknowledgments

We are grateful to Martin Rinard, Guy Tremblay, Nathan Tawil, Pedro Dinez, and the anonymous referees for their valuable comments. Further, we want to thank Guy Tremblay, Wim Bohm, Olaf Lubeck, Jeff Hammes, Christina Flood, R. Paul Johnson, and the whole Id-in-Id compiler group at MIT for providing us with the large collection of Id programs. Computational support at Berkeley was provided by the NSF Infrastructure Grant number CDA-8722788. Klaus Erik Schauser received research support from the Department of Computer Science at UCSB. Seth Copen Goldstein is supported by an AT&T Graduate Fellowship.

References

[AA89] Z. Ariola and Arvind. P-TAC: A parallel intermediate language. In *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*, pages 230–242, September 1989.

[AE88] Arvind and K. Ekanadham. Future Scientific Programming on Parallel Machines. *Journal of Parallel and Distributed Computing*, 5(5):460–493, October 1988.

[AHN88] Arvind, S. K. Heller, and R. S. Nikhil. Programming Generality and Parallel Computers. In *Proc. of the Fourth Int. Symp. on Biological and Artificial Intelligence Systems*, pages 255–286. ESCOM (Leider), Trento, Italy, September 1988.

[All92] L. Allison. Lazy dynamic-programming can be eager. *Info. Proc. Letters*, 43(4):207–212, 1992.

[All93] L. Allison. Application of recursively defined data structures. *Australian Comp. Journal*, 25(1):14–20, 1993.

[ANP87] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. Technical Report CSG Memo 269, MIT Lab for Comp. Sci., February 1987. (Also in *Proc. of the Graph Reduction Workshop*, Santa Fe, NM. October 1986.).

[ANP89] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[BCS+89] P. J. Burns, M. Christon, R. Schweitzer, O. M. Lubeck, H. J. Wasserman, M. L. Simmons, and D. V. Pryor. Vectorization of Monte-Carlo Particle Transport: An Architectural Study using the LANL Benchmark "Gamteb". In *Proc. Supercomputing '89*. IEEE Computer Society and ACM SIGARCH, New York, NY, November 1989.

[BH87] A. Bloss and P. Hudak. Path Semantics. In *Mathematical Foundations of Programming Language Semantics (LNCS 298)*. Springer-Verlag, April 1987.

[BH93] W. Bohm and R. E. Hiromoto. A Functional Implementation of the Jacobi Eigen-Solver. Tech. Report CS-93-106, Colorado State University, May 1993.

[Bir84] R. S. Bird. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica*, 21(4):239–250, 1984.

[BNA91] P. S. Barth, R. S. Nikhil, and Arvind. M-Structures; Extending a Parallel, Non-strict, Functional Language with State. In *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture*, Cambridge, MA, August 1991.

[CGSvE93] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. TAM — A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, 18:347–370, July 1993.

[CPJ85] C. Clack and S. L. Peyton-Jones. Strictness Analysis - A Practical Approach. In *Proc. Functional Programming Languages and Computer Architecture*, Sept. 1985. Springer-Verlag LNCS 201.

[CSS+91] D. Culler, A. Sah, K. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proc. of 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa-Clara, CA, April 1991.

[Fel91] M. Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1-3):35–75, December 1991.

[Gol94] S. C. Goldstein. Implementation of a Threaded Abstract Machine on Sequential and Multiprocessors. Master's thesis, Computer Science Division — EECS, U.C. Berkeley, 1994. (UCB/CSD 94-818).

[HA87] P. Hudak and S. Anderson. Pomset Interpretations of Parallel Functional Programs. In *Proc. Functional Programming Languages and Comp. Arch.*, September 1987.

[HCAA93] J. Hicks, D. Chiou, B. S. Ang, and Arvind. Performance Studies of Id on the Monsoon Dataflow System. *Journal of Parallel and Distributed Computing*, 18:273–300, July 1993.

[HDGS91] J. E. Hoch, D. M. Davenport, V. G. Grafe, and K. M. Steele. Compile-time Partitioning of a Non-strict Language into Sequential Threads. In *Proc. Symp. on Parallel and Distributed Processing*, Dec. 1991.

[Hel89] S. K. Heller. *Efficient Lazy Data-Structures on a Dataflow Machine*. PhD thesis, Dept. of EECS, MIT, February 1989.

[HFA+94] P. H. Hartel, M. Feeley, M. Alt, L. Augustsson, P. Baumann, M. Beemster, E. Chailloux, C. H. Flood, W. Grieskamp, J. H. G. van Groningen, K. Hammond, B. Hausman, M. Y. Ivory, P. Lee, X. Leroy, S. Loosemore, N. Röjemo, M. Serrano, J.-P. Talpin, J. Thackray, P. Weis, and P. Wentworth. Pseudoknot: a Float-Intensive benchmark for functional compilers (DRAFT). In J. R. W. Glauert, editor, *Implementation of Functional Languages*, pages 13.1–13.34. School of Information Systems, Univ. of East Anglia, Norwich NR4 7TJ, UK, Sep 1994.

[HL93] J. Hammes and O. Lubeck. MCNP-ID — A Dataflow Photon Transport Code. LANL, June 1993.

[Hug88a] J. Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):187–208, 1988.

[Hug88b] R. J. M. Hughes. *Backwards Analysis of Functional Programs*, pages 187–208. Elsevier Science Publishers, B.V. (North Holland), 1988.

[Joh87] T. Johnsson. Attribute Grammars as a Functional Programming Paradigm. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, February 1987. Springer-Verlag LNCS 274.

[Nik90] R. S. Nikhil. Id (Version 90.0) Reference Manual. Technical Report CSG Memo, MIT Lab for Comp. Sci., 1990.

[Nik91] R. S. Nikhil. The Parallel Programming Language Id and its Compilation for Parallel Machines. In *Proc. Workshop on Massive Parallelism, Amalfi, Italy, October 1989*. Academic Press, 1991.

[Nik93] R. S. Nikhil. An Overview of the Parallel Language Id (a foundation for pH, a parallel dialect of Haskell). Technical report, Digital Equipment Corp., Cambridge Research Laboratory, September 1993.

[Pey87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[Sah91] A. Sah. Parallel Language Support for Shared memory multiprocessors. Master's thesis, Computer Science Div., University of California at Berkeley, May 1991.

[SB93a] S. Sur and W. Bohm. Analysis of Non-Strict Functional Implementations of the Dongarra-Sorensen Eigensolver. Tech. Report CS-93-133, Colorado State University, December 1993.

[SB93b] S. Sur and W. Bohm. Efficient Declarative Programs: Experience in Implementing NAS Benchmark FT. Tech. Report CS-93-128, Colorado State University, October 1993.

[SB93c] S. Sur and W. Bohm. NAS parallel benchmark integer sort (IS) performance on MINT. Tech. Report CS-93-107, Colorado State University, May 1993.

[SCG95] K. E. Schauser, D. E. Culler, and S. C. Goldstein. Separation Constraint Partitioning — A New Algorithm for Partitioning Non-strict Programs into Sequential Threads. In *Proc. Principles of Programming Languages*, January 1995.

[Sch94] K. E. Schauser. *Compiling Lenient Languages for Parallel Asynchronous Execution*. PhD thesis, Computer Science Div., University of California at Berkeley, May 1994.

[SCvE91] K. E. Schauser, D. Culler, and T. von Eicken. Compiler-controlled Multithreading for Lenient Parallel Languages. In *Proc. Conf. on Functional Programming Languages and Comp. Arch.*, Aug. 1991.

[SGS+93] E. Spertus, S. C. Goldstein, K. E. Schauser, T. von Eicken, D. E. Culler, and W. J. Dally. Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5. In *Proc. of the 20th Int'l Symposium on Computer Architecture*, San Diego, CA, May 1993.

[SPR90] R. C. Sekar, S. Pawagi, and I.V. Ramakrishnan. Small domains spell fast strictness analysis. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 169–183, 1990.

[Tra86] K. R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report TR-370, MIT Lab for Comp. Sci., August 1986. (MS Thesis, Dept. of EECS, MIT).

[Tra91] K. R. Traub. *Implementation of Non-strict Functional Programming Languages*. MIT Press, 1991.

[Tre94] G. Tremblay. *Parallel Implementation of Lazy Functional Languages using Abstract Demand Propagation*. PhD thesis, McGill University, School of Computer Science, November 1994.