The Impact of Laziness on Parallelism and the Limits of Strictness Analysis

G. Tremblay

Dept. d'Informatique
Université du Québec à Montréal
C.P. 8888, Succ. Centre-ville
Montréal, Qué. H3C 3P8

G. R. Gao

School of Computer Science
McGill University
3480 University
Montréal, Qué. H3A 2K7

Abstract

The major question examined by this paper is whether sufficient fine-grain parallelism can be obtained from programs written in a lazy functional language. To answer this question, we have implemented a prototype compiler based on a novel approach to strictness analysis (called abstract demand propagation*) and we have compared this implementation strategy (optimized lazy) with other implementations, viz., pure lazy and lenient.*

Although the optimized lazy implementation improves significantly the efficiency of the resulting programs over the pure lazy approach, it was found that little parallelism can effectively be identified. This remains true even when a new notion of laziness — speculative laziness *— is introduced, notion well suited to parallel machines as it based on a* parallel *notion of* head-strictness *instead of the traditional sequential one.*

Our experiments also showed that when a program's result is known to be finite, then strictness analysis can generate almost as much parallelism as can be obtained from a lenient (i.e., non-strict but non-lazy) implementation. Thus, this means strictness analysis per se *is not sufficient and should be combined with some form of* termination analysis *and that there is little hope to extract much parallelism for programs that really require laziness.*

1 Introduction

1.1 Lazy functional programming languages

One of the major motivation for functional programming languages is as an alternative programming paradigm, especially for programming parallel machines [8, 25]. Functional programs are naturally parallel — the parallelism is *implicit*, based on the data dependencies, — and determinate — any order of evaluation always produces the same result.

There are three (3) major classes of functional languages which have been proposed, distinguished by their function invocation discipline:

- Strict: The arguments are evaluated *before* the function body is called (*call-by-value*).
- Lazy: The arguments are evaluated *only if* they are needed (*call-by-need*) and the resulting value is then memoized.
- Lenient: No *a priori* ordering is imposed on the evaluation of arguments vs. evaluation of the function body. The only constraint is that when an argument becomes needed, execution of the body must be suspended until the argument is available (*eager non-strict*) [41].

Lazy and lenient functional languages both provide non-strict behavior: a function can produce a result even when some of its arguments are not fully defined. [1] There are, however, some important difference between the two: in the lenient case, in order to always obtain a result, a fair scheduling strategy must be used and, because of the eagerness, some resources may be wasted; on the other hand, lazy evaluation ensures that no useless computation are ever performed, thus wasting no resources. Pragmatically, this makes it possible for lazy languages to define and manipulate data structures which are *potentially infi-*

[1] These notions are often confused. For example, all the programs presented in [9, 1, 2, 3] except one, contrary to what is claimed, require non-strictness, not laziness, as they all manipulate data structures which are finite.

nite, thus allowing elegant solutions to certain types of problems [27].

1.2 Lazy languages, parallelism and strictness analysis

As is well-known, the implementation of *lazy* languages requires much overhead [22]. Fortunately, it is possible to avoid part of this overhead by using a form of compile-time analysis called *strictness analysis*. Strictness analysis comes under various forms — forward abstract interpretation [13, 12], backward strictness analysis based on projections [29], rational strictness patterns [17], etc. — and is surely one of the most important factor for reducing the overhead incurred by lazy evaluation [19, 42].

For functional languages to be useful for programming parallel machines, it must be possible to extract sufficient parallelism, in order to utilize efficiently the resources of the underlying machine.

In the case of lenient languages used for scientific programming (i.e., programs dealing mainly with arrays and loops), it has been shown that ample fine-grain parallelism can effectively be extracted [36]. In fact, for these types of programs, too much parallelism can often be obtained, thus requiring some form of throttling mechanism [6, 23] in order to control the amount of parallelism and resources used. The objective of the present paper is to show that the situation is different for lazy programs.

1.3 Can lazy languages provide sufficient parallelism for fine-grain parallel machines?

The initial motivation that lead to the research described in the present paper was to give an answer, based on *experimental* evidences, to the question of whether lazy functional languages could provide sufficient parallelism to utilize efficiently the resources of a fine-grain parallel machine. This question is an important one given the motivations generally presented for the use of functional programming languages.

Of course, in order to obtain a meaningful answer, the lazy language implementation had to be based on a proper framework. In our mind, this meant a fully parallel framework where parallelism would be natural and could be expressed at all level, i.e., not simply a sequential framework extended with a few parallel constructs. Furthermore, it also meant a framework in which strictness analysis would be as precise as possible, especially for the analysis of dynamic data structures such as lists, trees, etc. These considerations lead the first author to the development of a novel (backward) approach for strictness analysis called *abstract demand propagation* [42] and to an implementation of this model within a compiler based on dataflow graphs.

Based on the compiler thus developed, a number of experiments have then been performed to answer the question mentioned earlier. These experiments are presented in Sections 4–6. As an *avant-goût* of what is to come, let's simply say that our conclusions are not optimistic: although strictness analysis can improve the performance of programs written in a lazy functional language, little parallelism can be extracted, even when attempts are made to go beyond the traditional (sequential) model of laziness. Furthermore, strictness analysis is not sufficient: programs really benefit from strictness analysis, in terms of parallelism, only when they are known *not to need* laziness — which clearly cannot be inferred from strictness analysis *per se*.

1.4 Outline of paper

The paper is organized as follows. Section 2 briefly presents the prototype compiler we have implemented based on our abstract demand propagation model of strictness analysis. Section 3 gives an overview of the experiments which were performed in order to determine the parallelism that could be extracted from lazy programs and hints at the major conclusions which were reached. The following sections then present these results in more detail. Section 4 shows how the traditional model of laziness seriously limits the exploitable parallelism, even when the optimizations allowed by strictness analysis result in improvement of the performance of the programs. Section 5 introduces some strategies that attempt to go beyond the traditional (sequential) model of laziness in order to improve the parallelism, while still preserving the ability to deal with potentially infinite structures. Section 6 motivates the need for other analysis beside strictness analysis, viz., termination analysis. Finally, Section 7 presents some related work while Section 8 concludes and indicates possible directions for future research.

2 A compiler based on abstract demand propagation

To examine the impact of laziness on parallelism, the first author developed a prototype compiler for a

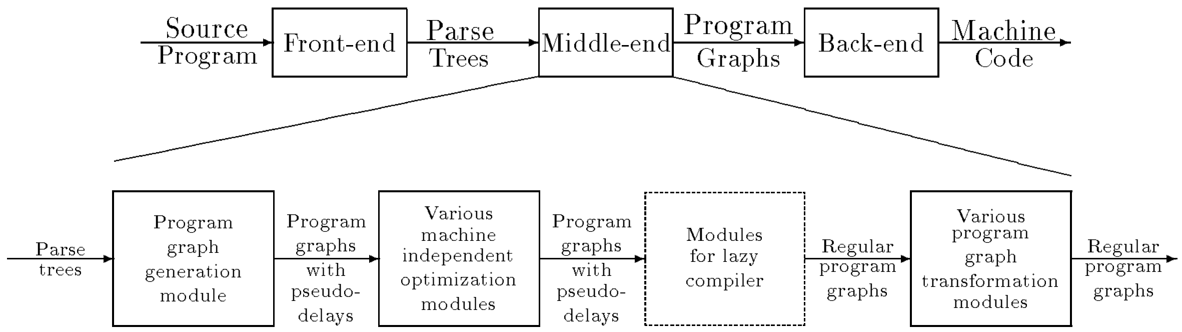

Figure 1: How the lazy compiler fit within the existing Id compiler

first-order purely functional subset of a lazy variant of the Id language [35], targeted for the TTDA dataflow architecture [7]. The lazy compiler was built using the existing Id compiler [40, 4], as shown in Fig. 1. Conceptually, an Id compiler consists of three components that manipulate distinct representations of the source program. The lazy compiler (dash box) consists in a number of modules (not shown in the figure) which are part of the middle-end; thus, all analysis and transformations performed by the lazy compiler are made at the program graphs level, so that it is fully independent of the target machine. The lazy compiler uses a *delay-and-force* style of implementation [22] based on the utilization of *L-structures* [21], structures which support an *implicit force-on-fetch* behavior for delayed expressions. The only (yet minor) modifications to the existing compiler were in the program graph generation module, where expressions occurring in possibly non-strict contexts were embedded within a new type of encapsulator (called *pseudo-delay*) in order to delay their evaluation. Of course, the *trick* is to avoid having such delays whenever possible, i.e., to remove them at compile-time instead of creating and forcing them at run-time. In our compiler, this is done using a form of strictness analysis called *abstract demand propagation*.

A detailed formal treatment of the approach of abstract demand propagation, which is beyond the scope of the present paper, would include both a discussion of the *soundness* of the approach — that there exists a well-defined and meaningful relationship between the notion of demand propagation and the standard semantics — and a discussion of its *safety* — although only approximate results are obtained, due to the use of an abstract interpretation, they do not lead to any unacceptable optimizations. Here, we only highlight some of the major characteristics of our approach; for a detailed presentation, see [42].

Intuitively, abstract demand propagation can be seen as a compile-time abstraction of the demand-driven evaluation process associated with lazy evaluation [15]. At compile-time, an initial demand is propagated backward through the program graph, in order to annotate the various delays. This process, which can also be seen as a form of *partial evaluation* [30, 32] relative to the demand for the output of the program, leads to the creation of multiple function versions, each one specialized for a specific demand. After all delays have been properly annotated (requiring a fixed point computation process in order to handle recursively defined functions), the delays that are associated with strict demands (i.e., those that are sure to be needed at run-time) can be removed, thus reducing the run-time overhead of the lazy evaluation process.

Although some forms of compile-time demand propagation have been used before to implement lazy languages [33, 38, 17, 39] our approach is novel in a number of ways:

- The notions of demands and demand propagation are given a formal interpretation in terms of the standard semantics through the definition of an exact (non-standard) demand propagation semantics and an associated abstract interpretation. Viewed in this way, demand propagation can be seen as an abstract way of computing the inverse of a program, i.e., determining the minimum inputs required by a program in order to produce at least the demanded output.

- The expressive power of the analysis made possible by our domain of demands allows interesting notions to be expressed, e.g., the fact that an argument is sure not to be needed, that an expression is sure not terminate, the notion of head-strictness [29] useful in the analysis of recursive data structures; furthermore, using a notion of *abstract conditions*, it can also handle cases of correlation between arguments, cases which cannot be handled by existing backward approaches.

As currently implemented, our analysis is thus similar, although slightly more precise and powerful, then what can be obtained using projection analysis [29], because our domain does not require demands to be uniform and allows demands to be of various sizes.

Below are two simple examples illustrating our demands. The first example is for the `len` function, which computes the length of a linear list:

```
Def len l =
   { Case l Of
      Nil = 0
   |  Cons x xs = 1 + len xs };
```

The result inferred under a positive scalar demand (denoted by \$) for `len` (i.e., `len` (\$)) would be the demand associated with `l` ($\mathtt{l} \Rightarrow$), as indicated below (1). This demand first says that the argument is strict, as "$T \rightarrow$" means the associated demand holds under condition $T =$ `True`, i.e., the demand is certain to be made; it also indicates that the associated demand is a *cyclic* demand (as indicated by "`Rec x::`"), i.e., a recursively defined demand. Here, `x` is associated with a demand indicating two possibilities — `Nil` or `Cons` — the second of which refers to `x` itself within its tail. Because this recursive reference in the tail is direct, i.e., is not delayed (cf. next example), it thus says that the structure (the *spine*) of the list must be fully evaluated. Furthermore, the head of the `Cons` demand indicates that none of the components (heads) are needed, as @ indicates a negative demand, i.e., sure *not to be needed*:

$$\begin{array}{l}\mathtt{len}\ (\$): \\ \quad \mathtt{l} \Rightarrow \{T \rightarrow (\mathtt{Rec\,x} :: \{\mathtt{Nil}, \mathtt{Cons}\,@\,\mathtt{x}\})\}\end{array} \tag{1}$$

Head-strictness — which allows the head of a cell to be evaluated as soon as the associated cell is created, even when not all cells are sure to be needed — can be illustrated with the following function, `before0`, which returns the elements that appear before the first 0:

```
Def before0 l =
   { Case l Of
      Nil = Nil
   |  Cons x xs = If x == 0
         Then Nil
         Else Cons x (before0 xs) };
```

The strictness result presented below (2) indicates that `before0` is head-strict under a head-strict demand (($\mathtt{Rec\,x} :: \{\mathtt{Nil}, \mathtt{Cons}\,\$\,\hat{}\mathtt{x}\}$)). Here, $\hat{}\mathtt{x}$ indicates an *uncertain* demand `x`, i.e., the element is not sure to be needed, but when it becomes needed, it will be needed at least with demand `x`.

$$\begin{array}{l}\mathtt{before0}\ (\mathtt{Rec\,x} :: \{\mathtt{Nil}, \mathtt{Cons}\,\$\,\hat{}\mathtt{x}\}): \\ \quad \mathtt{l} \Rightarrow \{T \rightarrow (\mathtt{Rec\,x} :: \{\mathtt{Nil}, \mathtt{Cons}\,\$\,\hat{}\mathtt{x}\})\}\end{array} \tag{2}$$

An important characteristic of our implementation, shared with the approach presented in [17, 18], is that it performs a process of *function specialization* [32, 14] based on the demand made for the result of the function being called. Each such new version is thus specialized to deal efficiently with a particular demand.

Various compile-time options were also defined, in order to investigate the factors that influenced the precision of the strictness analysis process. For lack of space, they will not be presented here; see [42, Chap. 5 and App. F] for a detailed presentation and examination of the influence of these various factors,

3 Overview of experiments and results

3.1 The benchmark programs

The benchmark programs that have been used to perform the experiments described in the following sections have been taken from a wide variety of sources: [22], [43], [27], [41], [21], [9], [3], [26], [37], [10], [44], [39], [20]. A brief description of these various benchmark programs can be found in Appendix A.

An important point to note concerning these benchmarks, however, is that they include both programs that can execute properly without laziness — i.e., that would behave and terminate properly using a strict or lenient implementation — *and programs that really require laziness* — i.e., that manipulate potentially infinite data structures. Such a distinction has not always been clearly made before; for example, [39] presents the result of a novel strictness analysis approach using only programs that would work perfectly well in a non-lazy implementation; in fact, in most cases, non-strictness is not even required. This distinction is important, as will be shown in Section 6, because these two types of programs should be handled differently.

3.2 The execution parameters measured

All our benchmarks were run on GITA [7], a dataflow machine simulator, and executed under an *idealized execution model*, i.e., assuming an infinite number of processors with no communication latency. Three key parameters have been measured and analyzed under this model:

- the length of the critical path (CPL), i.e., the minimum number of time steps required to execute the program;
- the total number of instructions executed (TOT);
- the average parallelism (AP), i.e., the average number of instructions executed per time step ($AP = TOT/CPL$).

In what follows, except in one case, we only present data relative to CPL, as data for TOT exhibit similar trends.

3.3 Summary of major results

Our first major result is that the optimizing lazy compiler can indeed improve the efficiency of the resulting program compared with a purely lazy implementation (i.e., no compile-time removal of delays). This is presented in Section 4, which also shows, however, that the optimization process based on strictness analysis has little impact on the resulting average parallelism. Thus, although the programs are more efficient, the average parallelism remains low, even when the programs do contain much more parallelism, as shown when executing them using the lenient approach.

Section 5 presents some attempts to improve the parallelism by going beyond the traditional model of laziness, which is inherently sequential. These attempts — the use of a parallel printing process, a new notion of head-strictness together with an appropriate notion of speculation — again improve the efficiency of the programs, but without any dramatic impact on the average parallelism.

Finally, Section 6 distinguishes between really lazy programs and programs that are known to produce and manipulate only finite objects, thus do not need the expressive power of laziness. It is seen that, in the latter case, it is possible to obtain almost as much parallelism as in the lenient (non-strict/non-lazy) case. Thus, this tends to indicate that strictness analysis *per se* is not sufficient and should be combined with some form of *termination* analysis.

4 The limits of parallelism for sequential laziness

The first experiment consisted in examining the impact of the optimizations — i.e., removal of unnecessary suspensions — made possible by our approach to strictness analysis. The results, for the critical path length (CPL), are presented in Fig. 2. This histogram indicates the number of benchmarks (horizontal axis) for which the reduction of CPL obtained by the optimizing lazy compiler (compared with the CPL of the pure lazy implementation) was in the indicated range (vertical axis). For instance, in 2 cases the optimizations lead to a reduction $> 1\%$ but $\leq 5\%$ of CPL, whereas in 42 cases this reduction was $> 20\%$. As was to be expected, the optimizations clearly have a positive impact on the efficiency of the resulting programs.

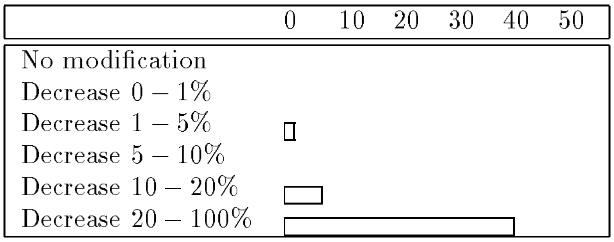

Figure 2: Pure vs. optimized lazy: Effect on critical path length (All benchmarks)

On the other hand, the optimizations have little impact on improving the average parallelism (AP), as can be seen in Figs. 3 and 4. For these histograms, each bucket indicates the number of benchmarks for which AP was in the indicated range. For instance, in Fig. 3, there were 39 cases where AP was ≤ 5, 6 cases where it was > 5 but ≤ 10, ..., and 2 cases where it was over 100.

Of course, an interesting question is whether this low average parallelism is due to laziness *per se* or to the inherent absence of parallelism within the selected benchmark programs. Our answer is that this is essentially due to laziness. This is illustrated in Figs. 5 and 6, where the optimized lazy vs. lenient (non-strict and non-lazy) implementations are compared and where it can be seen that more parallelism is available. Of course, in this case, only non-lazy benchmarks have been executed, i.e., programs that did not manipulate potentially infinite objects.

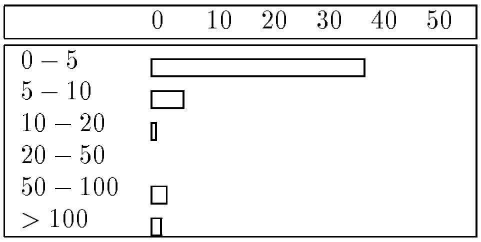

Figure 3: Average parallelism for pure lazy implementation (All benchmarks)

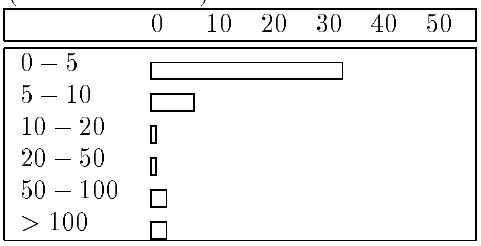

Figure 4: Average parallelism for optimized lazy implementation (All benchmarks)

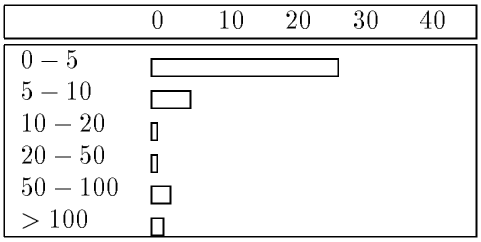

Figure 5: Average parallelism for optimized lazy implementation (Non-lazy benchmarks)

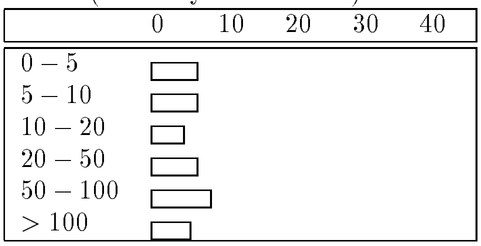

Figure 6: Average parallelism for the lenient implementation (Non-lazy benchmarks)

5 Beyond sequential laziness

In this section, a number of strategies are introduced as attempts to improve the parallelism: an alternative printing process and a new notion of head-strictness combined with an appropriate form of speculation. It will be seen that although the first strategy has a positive impact on the parallelism, its effect is not dramatic, whereas the impact of the latter one is negligible, even though it has a positive impact on the efficiency of the programs (reduction of CPL and TOT).

5.1 Sequential vs. parallel printing

The intent of the initial implementations of the lazy model of evaluation — both pure lazy and optimized lazy — was to emulate as faithfully as possible what occurs in a sequential lazy implementation. In such an implementation, printing is done using a (sequential) *left-to-right depth-first* traversal of the result object, where a component must be fully evaluated before evaluation of the next one is started. Clearly, such a printing process, although appropriate for a sequential implementation, is not well suited to a parallel machine. In the latter case, a *breadth-first* traversal would be more appropriate, where various components could get forced/evaluated in parallel.

Fig. 7 shows the effect, on CPL, of using a parallel printing process instead of a sequential one: as was to be expected, this generally reduces CPL. As was also to be expected (Fig. 8), this has hardly any effect on the total number of instructions executed: the same amount of work is done whether the traversal is done sequentially or in parallel, although this can be performed more rapidly in parallel.

When using a parallel printing process, because CPL is reduced more significantly than TOT, the average parallelism is slightly improved, as illustrated in Figs. 9 and 10.

5.2 Parallel head-strictness and speculation

An alternative notion of head-strictness

In a lazy implementation, where a program's result might be potentially infinite, it is the printing of the result which forces the evaluation of (some of) the delayed components. Of course, it is reasonable to expect that only a finite portion of the object will be forced/printed. In our model, the depth to which a

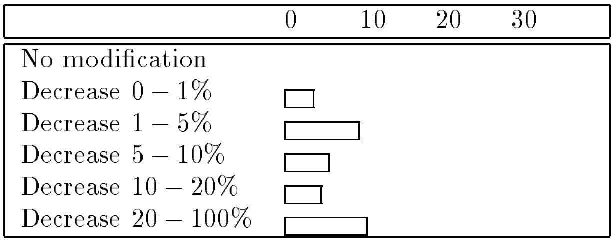

Figure 7: Sequential vs. parallel printing: Effect on critical path length

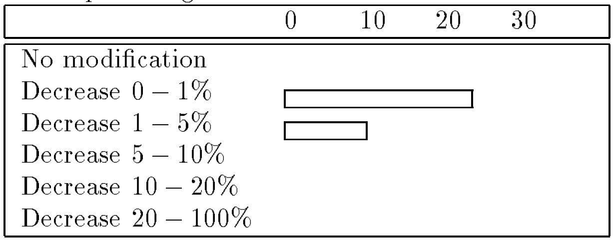

Figure 8: Sequential vs. parallel printing process: Effect on total number of instructions

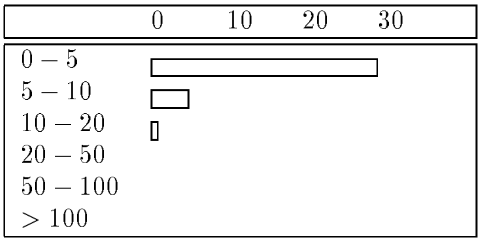

Figure 9: Sequential printing process: Average parallelism

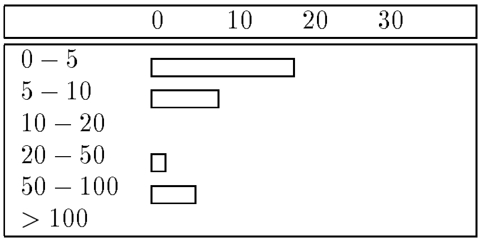

Figure 10: Parallel printing process: Average parallelism

program's result gets printed is determined by a run-time parameter called the *initial run-time demand.* [2]

On the other hand, the strictness analysis process, which is based on the (backward) propagation of demands through the program graphs, is triggered by a demand called the *initial compile-time demand*. This demand, in order to faithfully represent what might be demanded at run-time, must not be stronger than what can minimally demanded at run-time.

In our previous experiments, the initial compile-time demand was assumed to be a head-strict demand [29, 17]. Such a demand, which indicates that the outermost structure of an object can be evaluated together with its leftmost component, represents the minimal demand that can be made at run-time by a sequential lazy evaluator (performing, as mentioned above, a left-to-right depth-first traversal). In other words, a sequential head-strict demand ensures that no component of a structure S gets evaluated unless all components that precede it when sequentially printing S have been fully evaluated and printed.

However, now that a different printing process is used, a different notion of head-strictness can be used to represent the initial compile-time demand. This new notion can be illustrated by the example below. Suppose we are given the following `tree` type definition (where `N` denotes an integer component):

```
Type tree = Leaf2 N N | Node2 tree N N tree;
```

The appropriate sequential head-strict demand for this type would be the following, which delays everything to the right of the first component:

$$(\texttt{Rec}\ x\ ::\ \{\texttt{Leaf2}\,\$\,\hat{}\$,\ \texttt{Node2}\,x\,\hat{}\$\,\hat{}\$\,\hat{}x\})$$

On the other hand, the corresponding parallel head-strict demand would allow the non-recursive sibling components to be evaluated concurrently:

$$(\texttt{Rec}\ x\ ::\ \{\texttt{Leaf2}\,\$\,\$,\ \texttt{Node2}\,\hat{}x\,\$\,\$\,\hat{}x\})$$

Parallel head-strictness alone does not have any significant impact on performance or parallelism; in fact, there are cases for which a parallel head-strict demand can be weaker than a sequential head-strict one. Parallel head-strictness, however, leads naturally to a form of compile-time *speculation* that preserves the main property of lazy evaluation, viz., the ability to manipulate potentially infinite data structures.

[2] This parameter is an integer value and effectively indicates the number of levels down to which recursive data structures must be forced. The printing of an infinite object can then be seen as the limit of its printing as the run-time demand goes from 0 to ∞ [5]. In our experiments, the initial run-time demand was set to level 50, so that most programs producing finite results got fully evaluated.

Speculative laziness through parallel head-strict speculation

Any notion of *speculation* should impose some bound on the amount of speculative work. In our case, *speculative laziness*, we impose that no potentially infinite data structure should be fully constructed/ unfolded unless this would also occur in a lazy implementation.

In our approach, this is made possible by the notion of *parallel head-strict speculation*. As mentioned earlier, the intuitive idea behind parallel head-strictness is to emulate a breadth-first unfolding of data structures. When no speculation is allowed, all recursive components get delayed. On the other hand, when the speculation level is set to k, then whenever a delayed component gets forced up to k additional unfolding of recursive components can be triggered. For example, given the `tree` type definition presented earlier, the following demand would be obtained for a speculation level of 2:

$$(\texttt{Rec}\, x \,::\, \{\texttt{Leaf2}\,\$\,\$,\ \texttt{Node2}\, x'\,\$\,\$\, x'\})$$

$$\textbf{Where}\ x' = \{\texttt{Leaf2}\,\$\,\$,\ \texttt{Node2}\, x''\,\$\,\$\, x''\}$$

$$\textbf{Where}\ x'' = \{\texttt{Leaf2}\,\$\,\$,\ \texttt{Node2}\, \hat{}x\,\$\,\$\, \hat{}x\}$$

Thus, each time a new node gets forced, 2 additional levels of the tree may get created (if they exist).

Figs. 11–13 show the effect on CPL of increasing the speculation level, from, resp., 0 to 1, 1 to 2, and 2 to 3. It can be seen that increasing the speculation level leads in almost all cases to a slight reduction of CPL. [3]

The effect of speculation on the average parallelism is shown in Figs. 14–17. Increasing the speculation level has hardly any effect on the distribution of the average parallelism; in fact, as can be seen for a speculation level of 3, there are cases where AP is decreased. Thus, although the notion of speculation is intuitively appealing, it does not seem to bring much benefits, at least in terms of the parallelism that can be extracted from programs. [4]

[3] The exceptions are cases where the result is a potentially infinite structure. In this case, increasing the speculation level may mean producing *more* of the result, thus the increase for CPL (and TOT).

[4] The reason why only small levels of speculation (0–3) have been used is that our notion of speculation is a *compile-time* notion, not a run-time one. Thus, for certain types of data structures, e.g., trees, using a large speculation level leads to a significant increase in the compilation costs and rapidly becomes impractical. For example, dealing with binary trees requires demand structures whose size grow exponentially with the speculation level.

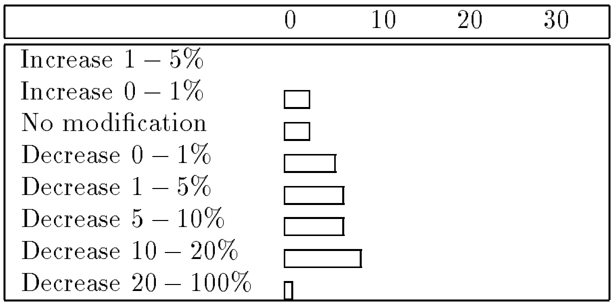

Figure 11: No speculation vs. speculation level = 1: Effect on critical path length

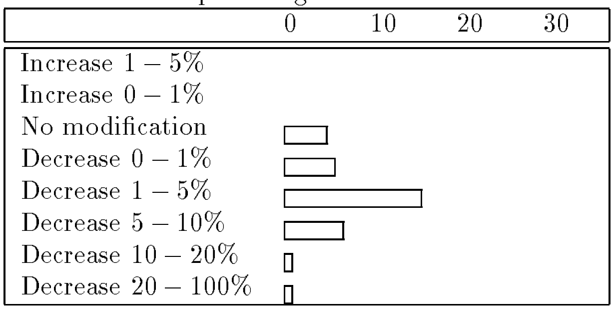

Figure 12: Speculation level = 1 vs. 2: Effect on critical path length

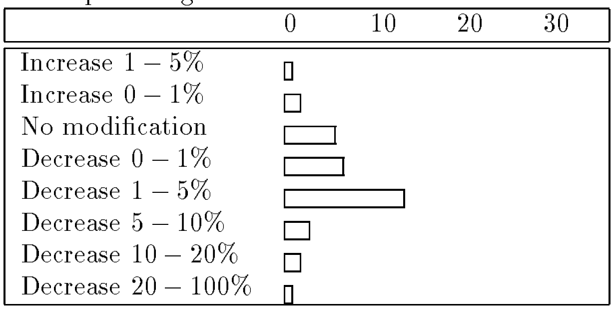

Figure 13: Speculation level = 2 vs. 3: Effect on critical path length

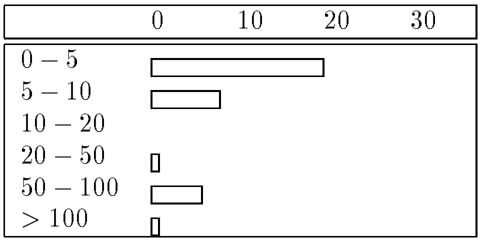

Figure 14: Speculation level = 0: Average parallelism distribution

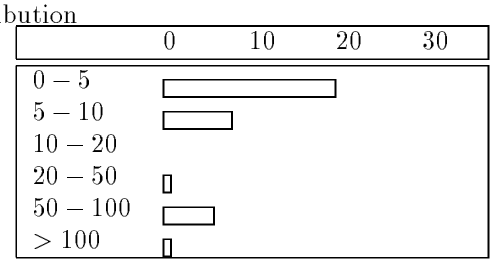

Figure 15: Speculation level = 1: Average parallelism distribution

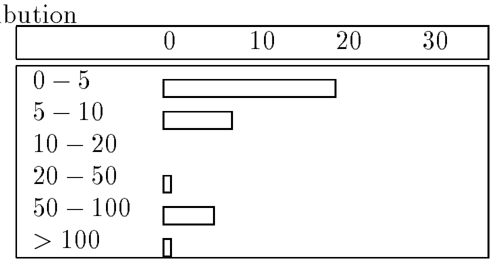

Figure 16: Speculation level = 2: Average parallelism distribution

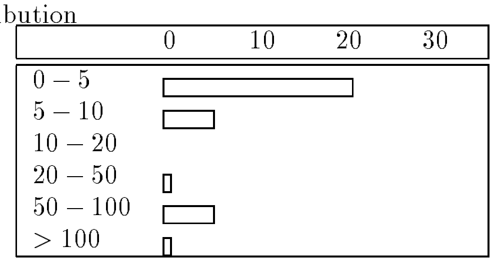

Figure 17: Speculation level = 3: Average parallelism distribution

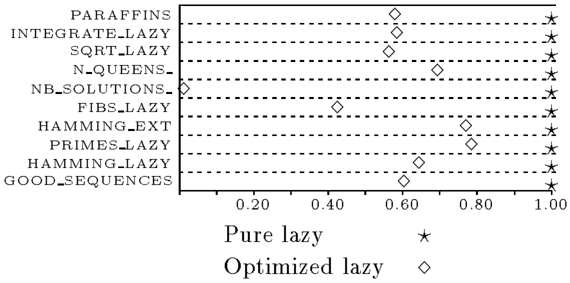

Figure 18: CPL for really lazy benchmarks relative to pure lazy implementation

6 Really lazy vs. non-lazy programs and the limits of strictness analysis

As mentioned earlier, our experiments were performed both with programs that required laziness and programs not requiring it. Section 6.1 examines the former ones — *really lazy* programs — i.e., those that manipulate data structures which are potentially infinite, whereas the latter are examined in Section 6.2.

6.1 Parallelism for really lazy programs

For really lazy programs, going from a pure lazy to an optimized lazy implementation has a significant effect on both CPL and TOT. This is illustrated in Fig. 18, where the CPL relative to the pure lazy implementation's CPL is indicated for each benchmark. For instance, the CPL of the optimized lazy implementation for `GOOD_SEQUENCES` was 60% of the CPL for the non-optimized implementation (i.e., the optimizations lead to a 40% reduction). However, as shown in Fig. 19 for the case with no speculation, the average parallelism is low, the only exception being for `NB_SOLUTIONS_`, which computes the length of the result list produced for `N_QUEENS_`, thus which effectively introduces an hyper-strict demand for that list (cf. Section 6.2). Similar results are obtained when the speculation level is increased.

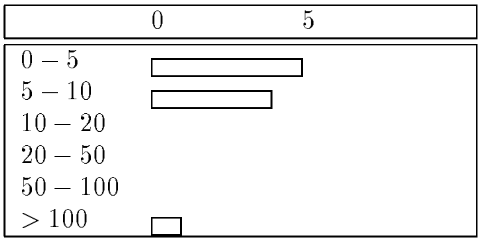

Figure 19: Average parallelism for really lazy benchmarks (optimized lazy)

6.2 Parallelism for non-lazy programs under hyper-strict demands

When a program produces a result which is potentially infinite, the only safe initial compile-time demand is some form of head-strict demand (sequential or parallel, with or without speculation); a *hyper-strict* demand, i.e., a demand asking for the structure to be fully evaluated, would not be acceptable because it could never satisfied. For example, such a demand would be equivalent to computing the length of an infinite list!

On the other hand, when a program produces a result known to be finite and the program is known not to manipulate potentially infinite data structures, then it would be perfectly safe to use a hyper-strict demand. Although this might perform more work then a pure lazy implementation, the amount of additional work would certainly be bounded in the sense described above.

Fig. 20 illustrates the effect on CPL of using hyper-strict demands instead of head-strict ones for programs not requiring laziness. In most cases, this reduces significantly the critical path length.

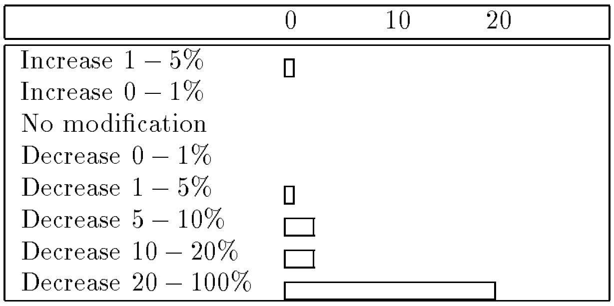

Figure 20: Effect on CPL of using hyper-strict initial demands instead of head-strict ones for non-lazy programs

What is more interesting is the impact on the distribution of the average parallelism, as presented in Figs. 21–23: the parallelism is effectively improved (Fig. 21 vs. 22); furthermore, the distribution becomes almost similar to the one for lenient evaluation (Fig. 22 vs. 23).

These results suggest that strictness analysis by itself is not sufficient to obtain ample parallelism for programs written in a lazy language: it would be useful for the compiler to know when a program does not need laziness. This could either be done through some form of annotations and/or some form of *termination analysis*, i.e., an analysis that would detect whether

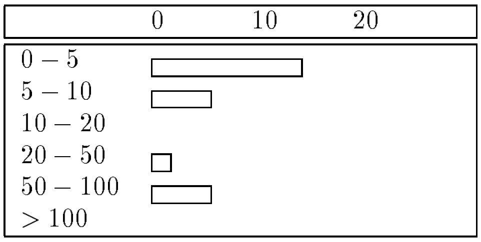

Figure 21: Average parallelism for optimized lazy implementation with an initial head-strict demand

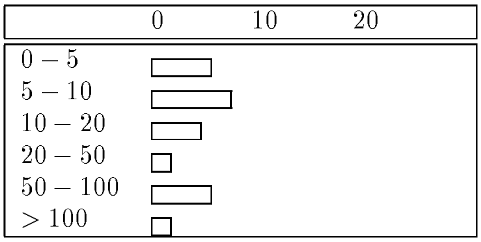

Figure 22: Average parallelism for optimized lazy implementation with an initial hyper-strict demand

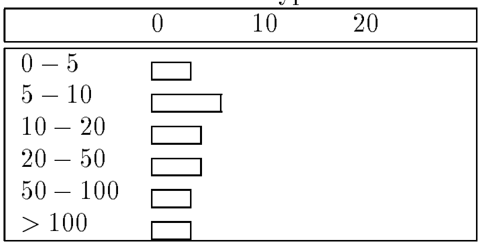

Figure 23: Average parallelism for lenient compiler

a program is sure to terminate under a hyper-strict demand.

7 Related work

In [24], a parallel implementation of a lazy language is described, where the identification of potential parallelism is done using the evaluation transformer model of [11, 12]. Because their target architecture does not support fine-grain parallelism, they only examine the effect of the tasks' size on the efficiency of the resulting programs. The potential parallelism that could be obtained on a fine-grain parallel machine is not examined.

The work of [31] describes a parallel implementation of a lazy language on a shared memory multiprocessing machine. The exploitable parallelism is identified using the analysis described in [39] — *dd-analysis* and *ee-analysis* — and the speedups obtained for various test programs are examined. They conclude their approach can identify all of the parallelism, i.e., they can always obtain almost linear speedup. However, there is an important weakness in this conclusion: *none of their program* really *require lazy evaluation*, that is, none exploit the major feature of lazy evaluation, namely, the ability to manipulate potentially infinite objects.

In [16], an assessment of the evaluation transformer model of reduction [11] on the performance of lazy programs is presented. Although various parameters are examined, e.g., heap allocation, number of memory accesses, etc., parallelism *per se* is not examined. Note that one of their interesting conclusion is that "it is worth investigating more complicated patterns of evaluation for nested data structures" [16, p. 334]; this is effectively made possible by the approach used in our prototype compiler, using a domain of (possibly) cyclic demands with a better expressive power than evaluation transformers.

Finally, [19] and [20] examine the impact of various compile-time analyses and implementation choices on the efficiency of a sequential implementation of a lazy functional language. Although a large number of factors are examined, the impact on parallelism is not examined because all the implementations being compared are sequential.

8 Conclusion

In this paper, we have shown that lazy languages, even when implemented using a framework well-suited for parallelism, can generate only small amount of parallelism. Although the types of programs used are different — recursive data structures and functions vs. arrays and loops — this is quite in contrast with what occurs for lenient languages, where too much parallelism can often be generated. So, whereas a lenient language might require a throttling mechanism to reduce the amount of parallelism, a lazy language might benefit from a mechanism to enhance parallelism. Speculative laziness, although such a tentative mechanism, did not have a significant effect, at least for the small levels of speculation used; some form of *run-time* instead of compile-time speculation might possibly have a more positive impact, which remains to be investigated.

The result of Section 6, which distinguished between really lazy and non lazy programs, can be interpreted in two ways. On one hand, they may point to the need for other forms of compile-time analysis, viz., termination analysis: when it is known that a program does not manipulate potentially infinite data structures, it is possible to obtain parallelism almost as high as for a lenient implementation. On the other hand, this result, when combined with the result concerning the low parallelism for really lazy programs, can also be interpreted as saying that there is not much hope for lazy languages as far as parallelism is concerned: in effect, they are saying that we can hope to obtain parallelism from programs written in a lazy language only when the programs *do not need* to be written in a lazy language!

This negative conclusion is reinforced by the fact that our programs were written in a first-order subset of Id, i.e., higher-order functions were not used. Strictness analysis of higher-order functions is a problem for which it is even more difficult to obtain good and precise results [28].

Although lazy functional languages are conceptually elegant and appealing, maybe it is time to reexamine the real usefulness of laziness — as opposed to non-strictness —, to determine the type of problems and applications for which laziness is really necessary. [5]

[5] As an anecdote, the first author, a year or so ago, made a request on the Internet (comp.lang.functional) for examples of lazy programs. Although *many* requests for the result of the survey were received, very few examples were sent ... and many of them did not even need laziness.

Acknowledgements

This research was partially funded by a National Science and Engineering Research Council (NSERC) grant. The first author would also like to acknowledge the support received from Université du Québec à Montréal (UQAM). Finally, we are also grateful to the anonymous referees for their comments. [6]

References

[1] L. Allison. Circular programs and self-referential structures. *Software — Practice and Experience*, 19(2):99–109, Feb. 1989.

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

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

[4] B.S. Ang, A. Caro, S. Glim, and A. Shaw. An introduction to the Id compiler. Computation Structures Group Memo 328, Laboratory for Computer Science, MIT, 1991.

[5] Z. Ariola and Arvind. A syntactic approach to program transformations. In *Symp. on Partial Evaluation and Semantics-Based Program Manipulation*, pages 116–129. ACM Press (SIGPLAN Notices), 1991.

[6] Arvind and D.E. Culler. Managing resources in a parallel machine. In J.V. Woods, editor, *Fifth Generation Computer Architecture*, pages 103–121. Elsevier Science Publishers, 1986.

[7] Arvind and R.S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. Computation Structures Group Memo 271, Laboratory for Computer Science, MIT, 1987.

[8] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *CACM*, 21(8):613–641, Aug. 1978.

[9] R.S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(4):239–250, 1984.

[10] A. Bloss, P. Hudak, and J. Young. Code optimizations for lazy evaluation. *Lisp and Symbolic Computation*, 1(2):147–164, 1988.

[11] G.L. Burn. Evaluation transformers — A model for the parallel evaluation of functional languages (extended abstract). In *Functional Prog. Lang. and Comp. Arch.*, pages 446–470. Springer-Verlag, LNCS-274, 1987.

[12] G.L. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation.* Research Monographs in Parallel and Distributed Computing. The MIT Press, 1991.

[13] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory of strictness analysis for higher order functions. In *Programs as Data Objects*, pages 42–62. Springer-Verlag, LNCS-217, 1985.

[14] K.D. Cooper, M.W. Hall, and K. Kennedy. Procedure cloning. In *Proc. 1992 Int'l Conf. on Comp. Lang.*, pages 96–105. IEEE Computer Society Press, 1992.

[15] A.L. Davis and R.M. Keller. Data flow program graphs. *IEEE Computer*, 15(2):26–41, Feb. 1982.

[16] S. Finne and G. Burn. Assessing the evaluation transformer model of reduction on the spineless G-machine. In *FPCA '93*, pages 331–340. ACM Press, 1993.

[17] C.V. Hall. *Strictness Analysis Applied to Programs with Lazy List Constructors.* PhD thesis, Dept. Of Computer Science, Indiana University, 1987.

[18] C.V. Hall and D.S. Wise. Generating function versions with rational strictness patterns. *Science of Computer Programming*, 12(1):39–74, June 1989.

[19] P.H. Hartel, H. Glaser, and J. Wild. On the benefits of different analyses in the compilation of a lazy functional language. In *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*, pages 123–145. University of Southampton, 1991.

[20] P.H. Hartel and K.G. Langendoen. Benchmarking implementations of lazy functional languages. In *FPCA '93*, pages 341–349. ACM Press, 1993.

[21] S.K. Heller. Efficient lazy data-structures on a dataflow machine. Technical Report MIT/LCS/TR-438, Laboratory for Computer Science, MIT, Feb. 1989.

[6] Which, among other things, made us reduce significantly the number of footnotes :-).

[22] P. Henderson. *Functional Programming — Application and Implementation.* Prentice-Hall International Series in Computer Science, 1980.

[23] 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, 1993.

[24] G. Hogen, A. Kindler, and R. Loogen. Automatic parallelization of lazy functional programs. In *ESOP '92 — European Symp. on Programming*, pages 254–268. Springer-Verlag, LNCS-582, 1992.

[25] P. Hudak. Conception, evolution, and application of functional programming languages. *Computing Surveys*, 21(3), Sept. 1989.

[26] P. Hudak and Anderson S. Haskell solution to the language session problem at the 1988 Salishan high-speed computing conference. Technical Report YALEU/DCS/RR–627, Dept. of Computer Science, Yale University, Jan. 1988.

[27] J. Hughes. Why functional programming matters. Programming Methodology Group Memo PMG-40, CTH Gothenburg, 1984.

[28] J. Hughes. Backwards analysis of functional programs. In *Partial Evaluation and Mixed Computation*, pages 187–208. IFIP/North-Holland, 1988.

[29] R.J.M. Hughes and P. Wadler. Projections for strictness analysis. In *Functional Prog. Lang. and Comp. Arch.*, pages 385–407. Springer-Verlag, LNCS-274, 1987.

[30] N.D. Jones. Automatic program specialization: A re-examination from first principles. In *Partial Evaluation and Mixed Computation*, pages 225–282. IFIP/North-Holland, 1988.

[31] O. Kaser, S. Pawagi, C.R. Ramakrishnan, I.V. Ramakrishnan, and R.C. Sekar. Fast parallel implementation of lazy languages – the EQUALS experience. In *Proc. of the 1992 ACM Conf. on Lisp and Funct. Prog.*, pages 335–344, 1992.

[32] J. Launchbury. Projections for specialisation. In *Partial Evaluation and Mixed Computation*, pages 299–315. IFIP/North-Holland, 1988.

[33] G. Lindstrom. Static evaluation of functional programs. In *Proc. of the SIGPLAN '86 Symp. on Compiler Construction*, pages 196–206, 1986.

[34] A. Maaβen. Parallel programming with data structures and higher order functions. *Science of Computer Programming*, 18(1):1–38, April 1992.

[35] R.S. Nikhil. Id (Version 88.0) reference manual. Computation Structures Group Memo 284, Laboratory for Computer Science, MIT, March 1988.

[36] R.S. Nikhil. The parallel programming language Id and its compilation for parallel machines. Computation Structures Group Memo 313, Laboratory for Computer Science, MIT, July 1990.

[37] R.S. Nikhil and Arvind. Id: A language with implicit parallelism. Computation Structures Group Memo 305, Laboratory for Computer Science, MIT, 1990.

[38] K. Pingali. *Demand-Driven Evaluation on Dataflow Machines.* PhD thesis, MIT, 1986.

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

[40] K.R. Traub. A compiler for the MIT tagged-token dataflow architecture. Master's thesis, Dept. of Electrical Eng. and Comp. Sci., MIT, 1986.

[41] K.R. Traub. Sequential implementation of lenient programming languages. Technical Report MIT/LCS/TR-417, Laboratory for Computer Science, MIT, 1988.

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

[43] D.A. Turner. The semantics elegance of applicative languages. In *Proc. of the 1981 Conf. on Funct. Prog. Lang. and Comp. Arch.*, pages 85–98, 1981.

[44] J. Young. *The Theory and Practice of Semantic Program Analysis for Higher-Order Functional Programming Languages.* PhD thesis, Yale University, May 1989.

A Benchmark programs description

The benchmark programs used for our experiments, which have been taken from a wide variety of sources,

Benchmark	Source	Description
PARAFFINS	[43]	Generates all paraffins.
INTEGRATE_LAZY	[27]	Integrates a function between two bounds using an infinite sequence of approximations.
SQRT_LAZY	[27]	Computes the square root of a number using an infinite sequence of approximations (Newton-Raphson's method).
N_QUEENS_	[21]	Generates all solutions to the N-queens problem using a list of all the solutions for all possible size.
NB_SOLUTIONS_	[21]	Computes the number of solutions obtained for N_QUEENS_.
FIBS_LAZY	[22]	Generates all Fibonacci numbers.
HAMMING_EXT	[26]	Generates all extended Hamming numbers.
PRIMES_LAZY	[22]	Generates all prime numbers using the sieve of Eratosthenes.
HAMMING_LAZY	[22]	Generates all Hamming numbers.
GOOD_SEQUENCES	[3]	Generates all irreducible (good) sequences over $\{1, \ldots, 5\}$.

Table 1: The really lazy benchmark programs

are presented in Tables 1 and 2. Note that, as mentioned earlier, we have included both programs that can execute properly without laziness (Table 2) and programs that really require laziness (Table 1). Also note that some of the programs have been used more than once in the experiments, using different inputs; for example, EQUAL_TREES counts for two cases: using, as input, two trees which are equal vs. two unequal ones. Finally, note that the programs for which the source has been left blank have been written by the first author.

Benchmark	Source	Description
N_QUEENS	[3]	Generates all solutions to the N-queens problem using circular trees.
NB_SOLUTIONS	[3]	Computes the number of solutions obtained for N_QUEENS.
PERMUTATIONS	[3]	Generates all permutations of a set of elements using a circular data structure.
HAMMING	[3]	Builds a Hamming sequence (using a circular structure).
REPLACE_WITH_MIN_		Creates a list with all elements replaced by the minimum value.
REPLACE_WITH_MIN	[9]	Same as REPLACE_WITH_MIN_, but using a circular list.
TRANSFORM_		Similar to REPLACE_WITH_MIN_, but for trees.
TRANSFORM	[9]	Like TRANSFORM_ but using a circular tree.
TRANSFORM_S_		Creates a tree with same shape as input, but with all elements sorted.
TRANSFORM_S_	[9]	Like TRANSFORM_S_, but using a circular tree.
FFT	[39]	Computation of the Fast Fourier Transform using arrays represented with lists.
REACHABLE_2	[34]	Computes all nodes reachable from a source node in a graph.
PRIMESTO_	[21]	Generates all primes up to a certain bound by checking against those already found.
PRIMESTO		Generates all primes upto a certain bound using the sieve of Eratosthenes.
COMPRESSPATH	[39]	Recursive doubling algorithm.
TRANSPOSE	[44]	Computes the transpose of a matrix (represented with lists of lists).
MM	[44]	Performs matrix multiplication.
WANG	[20]	Wang's algorithm for solving systems of linear equations based on large tri-diagonal matrix.
INTEGRATE	[39]	Numerical integration.
MYSQRT		Computes the square root of a number using a binary search.
FACT_LINEAR		Computes factorial using linear recursion.
FACT	[39]	Computes factorial using balanced divide-and-conquer approach.
FIB	[44]	Computes Fibonacci number.
MERGESORT	[39]	Sorts a list of numbers.
QUICKSORT	[39]	Sorts a list of numbers.
INSERTIONSORT	[21]	Sorts a list of numbers.
EQUALTREES_		Determines whether two trees are equal.
SAMEFRINGE	[21]	Determines whether two trees have the same fringe.
SIZETREE		Computes the number of nodes in a tree.
TREE6_S		Builds a tree.
SUMTREE		Computes the sum of the elements of a tree.

Table 2: The non-lazy benchmark programs