Improving the Performance of Parallel LISP by Compile Time Analysis

Jürgen Knopp
Siemens AG. Otto Hahn Ring 6
8000 München 83, Germany
email: jk@km21.zfe.siemens.de

Abstract

This paper presents a method to eliminate parts of the overhead introduced by parallel constructs in EDS LISP, a parallel extension of Common Lisp. *Futures* support a declarative style of explicit parallel programming. In languages with *futures* there is no static distinction between data calculated locally and data calculated by another process. Hence before data can be accessed a dynamic accessibility check has to be done. If the data is not yet *determined* an *implicit wait* has to be performed. Checks are done even when all data objects are present.
An algorithm is presented which does away with a significant amount of these checks. It is a variant of (intra-procedural) abstract interpretation, up to some extend similar to strictness analysis. Unlike other approaches the presented approach copes with both functional and non-functional LISP. The algorithm has been implemented in Common Lisp. The benchmarks are promising.

1 EDS LISP: Programming with Futures

The EDS Lisp (see [HaHe 90]) language supports explicit large grain parallelism. Aside from message passing and synchronization mechanisms the main language extension w.r.t. Common Lisp is the *future*. This construct is borrowed with some changes from other parallel Lisp dialects ([Hal 85]). We first show a (somewhat naive) parallel variant of *quicksort* using *futures*:

```
(DEFUN  QSORT (ARG-LIST)
(IF ARG-LIST                                                    ; (else NIL)
    (LET ((PIVOT (CAR ARG-LIST)) (REST-LIST (CDR ARG-LIST))
          (LEFT-LIST NIL) (RIGHT-LIST  NIL))
        (LOOP
            (IF REST-LIST
                (LET ((ACTUAL (CAR REST-LIST)))
                    (IF (> ACTUAL PIVOT)
                        (PUSH ACTUAL RIGHT-LIST)
                        (PUSH ACTUAL LEFT-LIST))
                (RETURN))                                       ;leave loop
            (SETQ REST-LIST (CDR REST-LIST)))
        (NCONC (QSORT LEFT-LIST) (LIST PIVOT) (FUTURE #'QSORT RIGHT-LIST)))))
```

The only difference to sequential quicksort is that the second recursive call is done in parallel using the *future* construct; left and right lists are sorted in parallel. To understand the semantics of *futures*, consider the call *(FUTURE function-name arguments)*. The *future* behaves almost like the Common Lisp construct *funcall*. The only difference is that it returns immediately - after the evaluation of the *arguments* - with some place-holder value. This place-holder stands for a result which is not yet computed. Eventually, the function denoted by *function-name* is applied asynchronously to the evaluated *arguments*.

When it returns the place-holder is overwritten with the real result. Place-holders are first class citizens : they may be copied and passed as argument to functions like all other values. Further, they may be used to build other values, e.g. as argument of the list-creating function *CONS*. Obviously, this black box treatment stops inside *primitive* constructs which "really need their arguments" like arithmetic operations. This is what we mean by *touching*. According to the semantics of *future*-based languages an *implicit wait* for the argument values has to be done in such circumstances: The evaluation is deferred until the argument value is accessible. In the above example the *implicit wait* is done for left lists but not for right lists. The reason for this is the fact that *NCONC* is not symmetric with respect to its arguments: the last one is handled in black box fashion.
There are two properties which are necessary for implicit waiting for the i'th argument of a construct call *(F... X...)*.

- the i'th argument position must be **touching**
- the construct *F* must be **primitive**

The properties with respect to touching result directly from the (sequential) semantics of predefined LISP constructs. In contrast, it is up to the language designer to classify constructs as *primitive* or *non-primitive* (with implications on the obtainable amount of parallelism).

2 Checks : The Price for Implicit Synchronization

To understand the aim of touching analysis it is necessary to have a look on some implementation aspects. As in sequential LISP objects are represented by tagged cells. *Future* objects are represented by cells with a special tag value. Further, a *future* cell contains a pointer to a data structure representing the "content" of the *future*. Hence there are 3 different kinds of objects to be considered:

- **undetermined future objects** can not be accessed immediately. The evaluation has to be suspended until the objects become *determined.*
- The values of **determined objects** can be read and written without any synchronization but they may have to be accessed through a chain of pointer dereferencing actions.
- **present objects** are conventional LISP objects. Of course, every *present* object is *determined.*

Given these three kinds of objects, *WAIT* (explicit and implicit) can be defined:

```
WAIT (X) =
if is-present (X) then X                          ;this check occurs very often
else                                                          ;and fails rarely
    if  future-is-not-computed (X) then dowait (X) endif   ;the evaluation is suspended
    WAIT (deref (X))                                        ; follow dereferencing chain
endif
```

The check in the first line is the point of interest. The aim its elimination. Provided we know at compile time that *X* is *present* the code simply becomes *WAIT (X) = X*. Less important, when we know that *X* is *determined* there is no need for

the check *future-is-not-computed* inside the recursion. In this case the check will fail for every recursion step and hence can be eliminated.

3 Touching Analysis : An Example

Consider the function *MAP-DESTRU* which performs a modification of its first argument (a list) by applying its second argument (a functional object) to each element of the list.
We first mark the touching positions with question marks.

```
(DEFUN MAP-DESTRU (SPINE ACTION)
  (LET ((ACTLIST SPINE)
        (ACTUAL NIL) (RESULT NIL))
       (LOOP
         (IF (? (NULL (? ACTLIST)))(RETURN))                    ;    leave loop
         (SETQ ACTUAL (CAR (? ACTLIST)))
         (RPLACA (? ACTLIST )(FUNCALL  (? ACTION) ACTUAL))
         (SETQ ACTLIST (CDR (? ACTLIST))))
       SPINE))
```

Obviously *CAR, NULL* and *CDR* touch their arguments. *IF* touches always its first argument, the touching of the second and third depending on the context (in this example, it does not touch). Note that *SETQ* (assignment) overwrites its first argument rather than touching it.
Applying the analysis we get some positions where we know that arguments are already *determined* (marked by "!") respectively *present* (marked by "$").

```
(DEFUN MAP-DESTRU (SPINE ACTION)
  (LET ((ACTLIST SPINE)
        (ACTUAL NIL) (RESULT NIL))
       (LOOP
         (IF ($ (NULL (? ACTLIST)))(RETURN))
         (SETQ ACTUAL (CAR (! ACTLIST)))
         (RPLACA (! ACTLIST )(FUNCALL  (? ACTION) ACTUAL))
         (SETQ ACTLIST (CDR (! ACTLIST))))
       SPINE))
```

Note the difference between *determined* and *present.*: the function *NULL* yields a *present* value. In contrast, it *touches* its argument making it *determined.*

Introducing Waits: The Assignment Paradigm

We can go one step further to obtain more *present* objects: for every argument on a touching position a *WAIT* will be performed anyhow. Why not perform the *WAIT* explicitly and overwrite the argument (in case it is a variable) with the result value ? (It is crucial to consider the exact definition of the semantics in order to allow such transformations. However, the rationale for this is beyond the scope of this paper .)

```
(DEFUN MAP-DESTRU (SPINE ACTION)
  (LET ((ACTLIST SPINE)
        (ACTUAL NIL) (RESULT NIL))
       (LOOP
         (IF ($ (NULL ($ (SETQ ACTLIST (WAIT (? ACTLIST))))) (RETURN))
         (SETQ ACTUAL (CAR ($ ACTLIST)))
         (RPLACA ($ ACTLIST )
                 (FUNCALL ($ (SETQ ACTION (WAIT (? ACTION)))) ACTUAL))
         (SETQ ACTLIST (CDR ($ ACTLIST))))
       SPINE)
```

We gain *present* values rather than *determined* ones. Note that there are more

present objects (and are indeed recognized by the analysis) than those marked but we do not bother about them because they do not occur on touching positions.

4 The Semantics of Touching Analysis

We now give somewhat simplified definitions for *touching*, *strictness*, *implicit wait*, *present*, *determined* and *primitive* in order to come to a more exact statement of the touching problem.

Every data object is uniquely determined by a **reference**.
Moreover, any object may have a **contents**. While the reference of an object exists during its lifetime the contents may be - at least initially - undefined. This is the typical situation when an object is created by a *future* function call.

Definition
An object is called **determined** iff it has a defined contents. A variable (or parameter) is *determined* iff its value is a *determined* object.

Definition
An object is called **present** iff it is *determined* during its entire lifetime. A variable is *present* iff its value is a *present* object. A variable or parameter is not *present* iff it is bound either to an object created by a *future* call or received its value from a variable or parameter which is not *present* either.
Obviously any *present* object variable or parameter is *determined.*

Note that both definitions ignore the structure of objects. *Present* or *determined* lists for instance may contain elements which have arbitrary properties.

Definition
A construct **application** $(F\ X_1 \ldots X_{i-1}\ Xi\ X_{i+1} \ldots X_n)$ is called **i-touching** iff for any Y (especially for any *undetermined* Y), Y **would** be *determined* after the application $(F\ X_1 \ldots X_{i-1}\ Y\ X_{i+1} \ldots X_n)$. In other words, the contents of the object resulting from the evaluation of the argument is needed during the function application. A construct F is called **i-touching** iff every application of F is *i-touching.*

Observation
In EDS LISP every *i-touching* construct is *i-strict.* The reverse is not true. In LISP all functions are strict (even functions like *CONS* !). Obviously, not all functions are touching (e.g. *CONS*).

Definition
A function application is called **i-result-touching** for some i iff its i'th result is *determined.* A function is called **i-result-touching** iff all its applications are *i-result-touching.*

The semantics definition of EDS LISP contains a specification of all *primitive* functions with respect to the properties *touching* and *result-touching.*

Examples
- the functions *CAR*, *CDR*, etc. are *touching* and not *result-touching.*
- +, /, -, etc. are *touching* w.r.t. all arguments and *result-touching.*
- the special form *SETQ* (assignment) is neither *touching* nor *result-touching.*
- the function *RPLACA* is *1-touching* and *result-touching.*

- the special form *IF* is *1-touching* and not *result-touching*.
- the functions *CONS* and *LIST* are not *touching*but *result-touching*.
- the function *FUTURE* is *1-touching* and not *result-touching*.
- the function *WAIT* is *touching* and *result-touching*.

Note that these properties yield in any context. Special calls like *(CAR (IF ...))* lead to touching of more arguments (of *IF*). But these are properties of the call context rather than properties of *IF*.

The semantics of EDS LISP with respect to implicit synchronization

- *WAIT* leads to synchronization iff its argument is not *determined*: The evaluation is suspended until its argument receives a value. This happens when the return value of a function spawned by a future becomes *determined*.
- The object returned by *WAIT* is *present* and has the same value as the argument of *WAIT*.
- For every touching position of a *primitive* construct an implicit *WAIT* occurs. No implicit *WAIT* occurs on other positions.

These assertions together with the touching property definitions of all *primitive* functions are part of the semantic definition for *EDS LISP*. Further, they are the basis of the touching analysis.

5 The Touching Analysis Scenario

Given a set of functions find those function applications (F ... X_i ...) which obey the following conditions:

a) F is *i-touching* and *primitive*.

b) X_i is *determined* or *present* (in the context where *(F ... Xi ...)* occurs).

The second condition holds in two typical situations:

1) X_i is touched by some other *touching* function or

2) X_i stems from a *result-touching* function application (including constants).

6 Abstract Interpretation

Abstract interpretation simulates the run time behaviour of programs in a simpler domain. For touching analysis this domain is three valued and totally ordered: ? < ! < $, in words: *no-prop* < *determined* < *present*. Properties of locals and parameters w.r.t. this domain are collected by the abstract interpretation algorithm. Worst case approximations are made for unknown data, conditionals and loops. It should be noted that there is no reason to restrict the analysis to purely functional programs. Loops and multiple assignments are handled in a straightforward manner. To obtain good results the analysis infers properties of user defined functions. This leads to intraprocedural optimization.

Recursion is handled in a way which is well known from strictness analysis:

- We start with an optimistic (!) assumption, namely that all arguments of the recursive calls are touched and all results are *determined* resp. *present.*
- The propagation is done iteratively. Every iteration computes new properties and new function bodies using the propagation algorithm of the non-recursive analysis for every step.
- The new property approximation is obtained by join between the old one and the properties obtained by the propagation.
- The iteration stops iff the function properties do not change any more.

Note that the start assumptions are optimistic rather than pessimistic. Although recursion and loops are treated very similar this is a fundamental difference.

7 Implementation Results and Related Work

The algorithm is implemented in Common LISP working by source to source transformation. It covers a functionality of over 100 LISP constructs including all aspects described in this document. Moreover *special* (dynamic) variables and *closures* are handled.
The identification of *present* with *determined* enforced by the assignment paradigm is optional. However, we feel that it is almost always useful.
Some benchmarks have been run before and after the analysis. We have counted the occurring *touching* positions and the *present* respective *determined* arguments on such positions. Both benchmark tables give ratios relative to the total

Check elimination ratios w.r.t. touching positions	1	2	3	4
without assignment paradigm : present's	0.17	0.16	0.14	0.59
without assignment paradigm : determined's	0.59	0.67	0.57	0.15
with introduced assignments : present's	0.67	0.83	0.71	0.84

1: MAP-DESTRU, 2: as MAP-DESTRU but with the first loop traversal unrolled,
3: non-destructive MAP, 4: DESTRUCTIVE benchmark (Gabriel benchmark)

number of touching positions. The rows in the first table show the elimination ratios for the analysis with and without introduced assignments, respectively.

Check elimination ratios w.r.t. touching positions	TAK *)	QSORT	FAC	FIB
non-recursive (naive) analysis	0.33	0.64	0.66	0.77
recursive analysis	0.60	0.67	0.79	0.84

Some well-known recursive functions (*): a Gabriel benchmark)

The second benchmark table shows results of the naive, non-recursive analysis versus the recursive analysis (the version with introduced assignments only).

All ratios show a significant improvement of the run time behavior (of course depending on the cost of these checks on an individual machine).

Touching analysis has been done in the Mul-T project in the context of a compiler for Scheme ([KrHaMo89]). The exact definition of the touching scenario, however, is lacking. Moreover, it seems that they analyze functional constructs, only. Although the touching analysis problem is different to the strictness analysis problem (see section 4), the method is very similar. With respect to the method we are very close to [ClPJo86]. However, there is no reason to restrict the method to functional programs.
Another related topic is type inference (see e.g. [PJo87], chapter 8 and 9).Touching analysis is a special case of type inference. Our algorithm is nicely extendible to type inference. The inference methods used in the context of functional languages do not fit for LISP. Note that LISP is neither functional nor type-safe. Hence properties are different on different program locations. This is in sharp contradiction to the concept of type in statically typed languages.
Methods used in some compiler optimizers are related to touching analysis, too. Lifetime analysis - albeit more simple - uses similar ideas (see e.g. [AhSeUl86]).

Due to page limitations, some of the aspects discussed in this paper are somewhat simplified. A more accurate framework with the formal justification w.r.t. recursive analysis for imperative languages is presented in a forthcoming paper. The extension to non-LISP *future-based* languages is subject of further work.

Acknowledgments
I wish to thank T. Henties, H. Ilmberger and M. Reich for fruitful discussions about strictness and touching and for proofreading earlier versions of this paper. This work was sponsored by the EC within project ESPRIT EP 2025.

References

[AbHa87] S. Abramsky, C. Hankin, editors: *Abstract Interpretation of Declarative Languages*, Ellis Horwood Series in Computers and Their Applications, 1987.

[AhSeUl86] A. Aho, R. Sethi, J. Ullmann, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[ClPJo86] C.Clack , S. Peyton-Jones: *Strictness Analysis: a Practical Approach*, in Proceedings of the ACM Conference on Functional Languages and Computer Architecture 1985, LNCS 201.

[HaHe 90] C. Hammer, T. Henties: *Parallel Lisp on a Distributed Machine*, in EUROPAL Workshop on High Performance and Parallel Computing in Lisp, London, 1990.

[Hal85] R. Halstead: *Multiplisp: A Language for Concurrent Symbolic Computation*, in ACM Transactions on Programming Languages and Systems, October 1985.

[KrHaMo89] D. Kranz, R. Halstead, Jr., E. Mohr: *Mul-T: A High-Performance Parallel Lisp*, in ACM Programming Language Design and Implementation , Portland, Oregon, June 1989.

[PJoCl87] S.Peyton-Jones, C.Clack: *Finding Fixpoints in Abstract Interpretation*, in [AbHa87].

[PJo87] S.Peyton-Jones, editor: *The Implementation of Functional Programming Languages*, Prentice Hall, 1987.