Reprinted from the

Proceedings of the GCC Developers' Summit

June 2nd–4th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Eric Christopher, *Red Hat, Inc.*
Janis Johnson, *IBM*
Toshi Morita, *Renesas Technologies*
Zack Weinberg, *CodeSourcery*
Al Stone, *Hewlett-Packard*
Richard Henderson, *Red Hat, Inc.*
Andrew Hutton, *Steamballoon, Inc.*
Gerald Pfeifer, *SuSE, GmbH*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Declarative world inspiration

Zdeněk Dvořák
SuSE Labs
dvorakz@suse.cz, http://atrey.karlin.mff.cuni.cz/~rakdver/

Abstract

The techniques for compilation and optimization of the declarative (logic and functional) programming languages are quite different from those used for procedural (imperative) languages, especially on the low level. There are however several reasons why they are still relevant even for the typically procedural language compilers like GCC: On higher level we can observe similarities, and due to more systematic design of the declarative languages the development in these areas is usually more advanced. In some contexts it is also considered a good style to use declarative programming techniques (recursion, generic programming, callbacks) even in imperative languages; currently the performance penalties for these constructs are usually quite large.

The paper quickly summarizes the similarities and differences between compilation of declarative and imperative languages. We then investigate the techniques used for declarative languages—tail recursion and general recursion optimizations, advanced inlining techniques (partial inlining, function specialisation, partial evaluation), program analysis, intermodular optimizations, etc., their usability and implementability in GCC.

Introduction

The contemporary programming languages can be divided into procedural and declarative. Programs in procedural languages describe precisely the control flow and they map more or less directly to the machine code of the target platform. On the other hand declarative languages focus on describing the semantics of the program. They do not describe that much how things should be done, but rather specify what result we would like to obtain, leaving the exact way how to do it up to the compiler. Of course this division is not all that clear—many procedural languages include some constructions derived from especially functional languages, and declarative languages usually contain procedural bits in order to handle things like input and output.

It is well-known fact that the compilation of declarative languages is in some sense both easier and harder than the compilation of procedural languages. Easier since the semantic description gives more freedom to the compiler and makes the analyzes simpler. Harder since the lack of explicit control flow makes it necessary to for a compiler to select a good order of execution by itself. This in general cannot be done in compile-time, so this makes it necessary to handle partially evaluated data in runtime. Also the more high-level nature of the declarative languages invites the programmers to use the constructions whose straightforward translation would be quite ineffective.

Of course on the low-level the techniques for compilation of procedural and declarative languages are quite different (it is also true that they also differ significantly between the various types of declarative languages). On higher-levels however the goals of the optimizations are more similar, and it is just here where the declarative languages may use the benefits of their cleaner semantics. Often it happens that even those high-level optimizations that could theoretically be used for procedural languages as well are only developed for the declarative ones, due to the problems with the level of analysis necessary to enable the alterations of the control flow prescribed by the procedural program. Also due to this diversity the research groups for declarative and procedural language compilation techniques do not communicate with each other frequently, so it may happen that the optimizations developed by one of them are either unknown or developed independently by the other one.

This paper tries to give an overview of (mostly high-level) optimizations used in declarative language compilers and to put them in context of the procedural language compiler GCC. We try to investigate their implementability and usefulness and also to derive some optimizations based on them that might be more useful for optimization of procedural languages.

First we provide a short introduction to the declarative language compiler construction and define the terms used in the area. Then we continue with the short descriptions of available optimizations, with more detailed descriptions for those that we consider relevant in the context of procedural languages. We also provide some thoughts and pointers on the eventual implementation in the current infrastructure of the GCC compiler (tree-ssa branch, since all the optimizations are only suitable for implementation on the tree level).

1 Compilation of Declarative Languages

In this section we provide a short introduction to the techniques used for compilation of the declarative languages. References to papers containing more detailed descriptions are provided. Of course the approaches different from the ones described below used as well, and the basic schemes can be altered to obtain variations useful for specific purposes.

We must distinguish between the different kinds of declarative languages, especially between logic (based on the predicate logic) and functional (based on the lambda calculus) ones. There are also other special purpose declarative languages (for constraint programming, database querying, scene description, etc.), but these are out of the scope of this paper.

Initial stages of compilation of all the languages, like lexical and syntactic analysis, are of course very similar and not interesting from our point of view. The optimizations (both generic and specific for the given style of the language) are then performed (some of them will be mentioned in the following sections). Usually the level of the representation is lowered during the process, finally leaving us with just the basic elements of the language. Type (and for logic languages mode—determining which arguments of predicates are input/output) checking and eventually specialization of operations happens during this process

For logic languages the basic elements are unification (that includes both construction and decomposition of data structures) and definition of predicates (that usually are recursive and use some built-in predicates for performing things like arithmetics). The language specification also defines the rule for order of evaluation of the predicates, which may be fixed (Prolog), flexible subject to some minimal constraints (Mercury) or even alterable by the program (Goedel). The possible substitutions to the variables are processed according to this rule, backtracking whenever such a substitution fails.

This rule (or its particular variant chosen by the compiler) is then translated into a code of a low-level abstract machine (which is later mapped to the target language). One of those used is the Warren Abstract Machine ([W83]). It consists of the low-level instructions to control unification, predicate lookup and backtracking. Indeed the main challenge at the low level is to make these operations efficient.

For unification it is necessary to handle the special cases of unification with terms with known structure, and to employ efficient algorithm for matching the terms with unknown structure when this fails. This is complicated by the fact that unification is two-way process (i.e. both unified sides may get modified). Also we need to be careful about the possibility to create the cyclic structures when compiling unifications like $X = f(X)$.

Predicate lookup is usually made faster by filtering out predicates that cannot match due to the known structures of parameters; this indexing may be either shallow (only looking at the topmost level) or deep. The things get more complicated in languages like Prolog where the program may be changed dynamically.

For backtracking we need to implement the rollback mechanism, either using timestamps or a clever layout of allocated data structures (or both).

For more details on construction of logic language compilers see for example [R94], [DC01] or [HS02].

For functional languages the basic elements are pattern matching (data structure decomposition), data structure construction and function application. Local function definitions are usually replaced by the global ones, in process called lambda-lifting. The functional languages often allow polymorphic functions (whose arguments may be of different types, similar to mechanism of virtual methods in object oriented programming); these are usually lowered to explicitly passing the dictionaries of the functions.

There are two commonly used semantics for functional languages regarding the passing of the arguments to the functions. The eager evaluation (Scheme, Erlang) means that the arguments are evaluated before they are passed to the function. The lazy evaluation (Haskell) means that they are only evaluated on demand, when the called function needs to know their values. The later approach is theoretically more clean (making the identities like $(\lambda\, x.f)g = f[x := g]$ valid even in cases when evaluation of g does not have to terminate), but significantly less efficient to implement (it is necessary to create thunks for unevaluated expressions whenever we pass an argument to a function) and the actual control flow is hard to predict, making the programs difficult to optimize. Nevertheless the methods of compilation of these languages are similar—even eager languages must be able to suspend evaluation of expressions when partially applied functions are passed as arguments, although their advantage is that from the type information they can often derive whether this occurs.

The examples of low-level abstract machines used for compilation of the functional languages are for example G-machine ([A84], [J84]) or the Three Instruction Machine ([FW87]). Despite the significant differences, the basic operations include manipulation and querying of the data structures (to enable their construction and pattern-matching), the parameter passing and the function calls.

There are two basic models used to call functions. The "eval-apply" model works by evaluating the function, in case of eager languages evaluating the arguments and applying the functions to the arguments. The "push-enter" model is used for lazy languages; it pushes the arguments of the function to the parameter stack, then enters evaluation of the function. There is no return after the end of the function in this case.

For lazy languages, it is necessary to ensure sharing. For example in $(\lambda\, x.x + x)\, f$ we want f to be evaluated just once. This means that when we finish evaluation of a thunk, we need to arrange its value to be rewritten by the result.

For the reasonable performance, there are several problems to be solved. We need to arrange for a sane argument/return value passing conventions using registers, and to make this work together with the argument stack. The partially applied functions present in the form of the thunks must have a mechanism how to apply additional arguments to them (by copying the thunk, creating the linked lists of arguments, or combination of both depending on the size of the thunk). We need to distinguish between already evaluated values and thunks, which may be done either by tagging or by keeping even evaluated values as trivial thunks that just return their value. Similarly either tagging or selector function needs to be used for distinguishing the variants of values during pattern matching.

For more involved description of these decisions as well as other issues with compilation of (especially lazy) functional languages see e.g. [J92] or [JL92].

Usually either some low-level procedural language (C) or assembler is used as the target language. The former is more portable and usually produces a better code due to the low-level target specific optimizations done by the C compiler, assuming that there is a possibility to ensure that the important values (for example stack and heap boundaries) are kept in registers all the time. The later is more complicated, but it gives a better compilation speed.

Of course these are just the basic approaches, which need to be enhanced by various low-level optimizations both on the source and the resulting code. In result, the performance of the more practical languages (functional with eager evaluation, logic without implicit backtracking) is in general the same as of the higher-level procedural languages. The performance of the languages that more precisely match the clean theoretical ideas (functional languages with lazy evaluation) tends to be worse by a factor of 2–5.

2 Declarative Language Optimizations

Many of the optimizations in the declarative languages try to eliminate the inefficiencies of the models described above. We omit the description of the low-level optimizations completely, since they clearly are not relevant, and also require a detailed knowledge of the particular model. The more high-level examples include

- Deforestation ([W90], [G96]) attempts to eliminate the need for creating temporary structures in clean declarative languages, where by clean we mean that the functions cannot have side effects, and consequently it is impossible to rewrite the data in-place (i.e. when you need to modify something, you must create its copy). This is remotely similar to loop fusion, although the main effect we want to obtain by it is quite different. For example programmer would usually write $map\, f\, (map\, g\, l)$ to apply two functions f and g to the list l. This however requires creation of the temporary list for the result of $map\, g\, l$. Deforestation rewrites this to $map\, (f\, .\, g)\, l$, which produces the result directly.

- In lazy languages, strictness analysis determines whether the arguments of the function will always be evaluated. If this is true, we may evaluate them directly and we do not have to create thunks for them.

- In logic languages, analysis of whether the predicate is deterministic (i.e. always giving just a single solution) can be used to omit the code necessary to handle backtracking.

Note that despite of the fact that the mentioned optimizations are quite high-level and they require nontrivial analyzes, they are obviously specific for the particular family of models and they do not seem to be directly applicable to the procedural programming languages (possibly with the exception of the limited version of deforestation in languages including map-like commands, but usually this can be handled by loop fusion as well).

Still there are some optimizations that seem relevant. The following sections are dedicated to them.

3 Recursion Elimination

Since the declarative languages do not in general include loop-like statements, all such constructions are achieved using recursion. Therefore it is important to handle recursion efficiently, and replace it by standard iterative loops as possible.

The simplest case is the tail call elimination (replacing the calls after that the function exits immediately by ordinary jumps). This optimization is standard in procedural languages as well, so we will not describe it in detail here. Instead we focus on some useful improvements to this basic scheme (most of them based on [LS99]):

- Provided that we have sufficient knowledge about the operations done after the call, we may be able to reorganize the computations and remove the recursion. Consider for example

```
f(x): if x == 0 then
        return 1;
      else if x % 2 then
        return 5 * f (x - 1);
      else
        return 3 + f (x - 2);
```

  This can be transformed into

```
f(x): m = 1;
      a = 0;

      start:
      if x == 0 then
        return m + a;
      else if x % 2 then
        {
          x--;
          m *= 5;
          goto start;
        }
      else
        {
          x -= 2;
          a += 3 * m;
          goto start;
        }
```

  This is what we currently do in GCC. Note that to get this result we needed a plenty of knowledge about nature of the operations $+$ and $*$—distributive law, associativity, commutativity, and existence of neutral elements. In the special case when just a single such operation is used and all non-recursive exits return the same value, associativity (and in some cases commutativity) would be sufficient, but even these are quite hard to check and this restricts this approach to just a limited class of programs.

- Provided that we have a sufficient knowledge about the operations done before the call, we may turn the recursion into iteration without changing the order of operations executed, in this way:

```
f(x): if x <= 0 then
        return 1;
      else
        return g (x, f(x-1));
```

into

```
f(x): if x <= 0 then
        return 1;
      r = 1;
      for (ax = 0; ax != x; )
        {
          ax++;
          r = g (ax, r);
        }
      return r;
```

We need the function to be in somewhat restricted shape to perform this transformation (see [LS99] for details, most importantly no unhandled code can be executed before the recursive call), the increment ($x \leftarrow x - 1$) needs to be invertible (see [HK92] for some theory on the topic; in practice probably just the simple induction variable-like increments could be handled), and we need to be able to determine the start value of the counter. On the other hand effects of g (or whatever code might be there) are unrestricted, since we do not change the order of execution of the calls to g.

- The situation becomes more complicated when one of the conditions above is not satisfied, but still sometimes it can be handled. For example if there are more exits and some code executed before the recursive call, we can still optimize the function by creating two loops—one executing the stuff done before the call and coming all the way down to the appropriate exit case, the second one identical to the one described in the previous case. This requires that those two pieces of code do not communicate with each other except for the value of the counter.

- Finally if there are also multiple recursive calls or we are unable to derive the inverse of the increment, we may eliminate the recursion by maintaining the stack ourselves. This gives less benefits than the previous cases, but still we only need to save variables that are live across the call, we save on the cost of the call itself (including parameter passing) and we expose the loops to the loop optimization (but see also the following section regarding the subject).

4 Loop Optimizations

As mentioned in the previous sections, loops in the declarative programming languages are almost exclusively expressed through recursion. Although we have demonstrated several powerful techniques for eliminating the recursion, in fact in many cases these approaches fail. It is therefore useful to be able to optimize such loops.

The interprocedural loops can be detected using the standard algorithms applied to the graph obtained by taking union of a callgraph and the control flow graphs of the functions. Since the strongly connected components of mutually recursive functions are usually entered at one point, the concept of the natural loop seems to be sufficiently general to cover the most important cases. On a side note, considering the ordinary intraprocedural loops in context of this graph may be useful as well, for example in order to be able to estimate instruction and data cache effects.

For the interprocedural loops the invariant motion and redundancy elimination seem to be the easiest to apply and the most useful from the standard optimizations (some other like strength reduction could work as well, but only under assumptions that are quite unlikely to happen). The implementation is straightforward:

- Determine the parameters and global variables that are just passed through unchanged, and propagate the information to determine those that are invariant.
- Run the function local invariant analysis.
- Move the computation of the invariants out of the loop. It may be necessary to create a wrapper around the header function of the loop (which is analogical to creating the preheaders) unless it is called from only one place outside of the loop.

If the moved invariants are expensive, we can create a global variable for them, since the loads from the memory will still pay up. Otherwise we must be able to reserve a register for them across the functions (which should be possible in GCC with just minor modifications). Obviously we must be very careful about the register pressure in this case.

On intraprocedural level, there are other interesting high-level loop optimizations. For example incrementalization (usage of the values computed in the previous iterations—see [LSLR02]) can be used to transform code like

```
for (i = 0; i < 100; i++)
  {
    sum[i] = 0;
    for (j = 0; j < i; j++)
      sum[i] += a[j];
  }
```

into

```
sum[0] = 0;
for (i = 1; i < 100; i++)
  sum[i] = sum[i - 1] + a[i - 1];
```

thus achieving an asymptotic speedup.

5 Inlining and Specialization

The declarative programs tend to be composed of small functions. To make the intraprocedural optimizations useful, it is necessary to perform function inlining intensively. See for example [JM02] for discussion of applicability and problems connected with inlining in lazy functional languages.

Also generic functions and usage of callback-type functions is a norm in these languages. They obviously carry a significant penalties for their usage with them—such functions are harder to optimize and often require passing of a partially applied function arguments or function dictionaries, which is not cheap. To overcome this, function specialization (also called cloning) is necessary. This optimization consists of creating duplicates of functions depending on the call site and optimizing them for the particular values or types of arguments. See for example [FPP00] and [P97] for more details.

Both of these optimizations are studied in the context of procedural languages as well, and at least some of the implementation issues should be covered by Hubicka ([H04]), so we make just a few minor points here.

- It is necessary to interleave inlining and specialization with other optimizations. The approach that tries to get the best performance would have to at least optimize the functions locally (to get rid of unreachable calls, decrease the function sizes and propagate the information about values of arguments), then inline/specialize the optimized function bodies, rerun the optimizations, then run inlining and specialization again (to exploit the interprocedural information taken into account due to the first inlining pass), then again rerun the optimizations. Of course this may get compile-time expensive, so other variations of the scheme may be useful at the lower optimization levels.

- The code growth is the major problem with both of these optimizations, since it has bad effects on the instruction caches. To overcome the problems, having a callgraph with profiling information is very useful—we then may optimize just the intensively used functions and function call sites.

  An implementability note: in fact we have basically everything needed in GCC with the current profiling scheme—it would be sufficient to tag the call sites in a unique way and to emit the call $\mapsto$ basic block map before profiling (similar to the current `.gcno` files). The other possibility would be the early instrumentation of the call sites. The main problem currently is that both of these possibilities interfere with the ordinary profiling. The former possibility needs the function inlining not to be run in the training pass. The later needs to be done before inlining and changes the code, so it cannot be done simultaneously with the ordinary instrumentation that is done after inlining. One of the solutions is to do the both at the same time, which again needs the inlining to be done later in the compilation process.

- Other possibility is to inline just the relevant parts of the function (so-called partial inlining). If we identify that there is a short hot part in the inlined function, we may copy just this part and put the rest into a separate shared function. This is useful especially for functions that cache their results, or handle common special cases in advance.

- There are several approaches to limit the code growth with specialization. One of them is to first specialize all possible occurrences, optimize the bodies and then reshare those for that we were not able to improve the code sufficiently. The other one is to identify applicability of optimizations in advance and just specialize those for that we believe it will be useful.

  None of these approaches seems to be suitable for GCC. The former obviously wastes a lot of compile time, and detecting the non-improved instances also would not be straightforward. The later is difficult to implement (it would need to have a separate analysis for each optimization) and unreliable. The realistic approach seems to identify the obvious possibilities (functions with callback arguments, boolean flags passed to them and guarding parts of the code in their bodies, constant integer parameters used as bounds of the loops, for example) and specialize just these. Additionally attribute mechanism could be used to give the programmer a possibility to tell that he wants the function to be specialized for the specified arguments.

- Interprocedural loop optimizations mentioned in the previous section as well as other optimizations may need to create simple wrappers around the functions. This may be useful in other cases as well. For example we do changing of calling conventions for static functions. If we detect that an exported function is often called locally (from the callgraph profiling, or just by determining that it is called recursively), creating an exported wrapper just calling its local instance may pay up. The other possibility would be to clone a local copy of it, but this would usually grow the code much more.

- Specialization on the constant arguments and specialization on types of arguments is the most commonly used option. Other possibility is to specialize according to the information from value range propagation or other analyzes, but currently there is not the infrastructure necessary to exploit this possibility in GCC.

6 Data Structure Analysis and Optimizations

This section describes some optimizations related to data structures used by the programs. They are mostly relevant for higher-level languages. Applying them for low-level procedural languages like C is complicated by the following issues:

- The layout of data structures is precisely given in C (by ABI for the particular architecture). This makes it only possible to alter it in cases when we are able to prove that there are no external references to the structure, and that the program does not rely on a particular layout of the data structure.

- The exposed pointer arithmetics makes all analyzes close to impossible. It is not easy to handle even the basic prerequisite for all the optimizations—alias analysis—in a satisfactory way.

Despite of these problems, some of the optimizations have also been studied in the context of procedural languages, since the memory access times are a bottleneck in many applications. For these reasons we provide only a short descriptions of several chosen optimizations, with references to relevant papers:

- Array reshaping changes the layout of arrays (order of indices and their dimensions) to improve the effectiveness of caches. See for example [G00] that implements the array padding (changing dimensions of an array by adding unused elements). Memory layout optimizations can be with advantage used together with loop nest optimizations ([CL95]).

- Linked lists are the basic structures used in the declarative languages. Therefore much of effort is directed to their optimizations. Although the procedural programs often also work with linked lists, usability of the techniques mentioned below is quite limited due to problems with identifying this pattern (see [CAZ99] for overview of such an analysis).

  If we are able to detect usage of linked lists, we may use the knowledge in several ways. We may arrange the memory to be allocated sequentially, thus improving cache behavior ([LA02] does a similar optimization, but without trying to identify precisely the access pattern). In cases when the list is accessed in a queue-like fashion only, we may also change the representation of the list, for example by putting several consecutive elements of the list to an array. This decreases the amount of memory needed by eliminating the successor (and possibly predecessor) pointers, allows more effective traversal of the lists (in loops controlled by a normal induction variable) and consequently increases efficiency of loop optimizations.

- Declarative languages often support use of temporary data structures (especially linked lists) in an almost transparent fashion, leading to initially quite ineffective code. This makes optimizations like dead store elimination for partially dead data structures necessary; see for example ([L98]).

- For some of these optimizations a memory access profiling might be useful. In the most expensive variant, the full list of all memory accesses tagged with the corresponding references to the source program and perhaps also exact values of indices for array accesses is recorded in the training pass. This data together with a memory cache model provides a quite exact base for determining the parameters for all cache directed optimizations. The optimizations that require exact analysis of the access pattern of course cannot be based just on this empiric data, but they may at least use it to locate the opportunities and to evaluate their usefulness.

  Obviously recording all memory accesses may turn quite expensive, so recording just the relevant information may be necessary. For implementation details in GCC see for example the recent works of the author and Caroline Tice on profiling driven array prefetching.

Conclusions

Several of the techniques we have presented appear to be implementable GCC (note that at least for some of them this would not be a simple task at all, however) and useful enough so that they might bring measurable speedups, especially

- improvements of the recursion elimination
- data access profiling and data structure reorganization
- call graph profiling
- function cloning and specialization

There are other optimizations that seem to be "cool" and implementable in the GCC framework, although they are only applicable in very special cases. They probably would not improve the performance much by themselves, but implementing them might be interesting from theoretical reasons. In some cases there also seem to be a chance to generalize them and thus improve their applicability. They include

- interprocedural loop optimizations
- loop incrementalization
- linear structures analysis and related optimizations

Of course there also are many optimizations that probably are only useful in context of declarative languages (deforestation, strictness analysis and unboxing, etc.)

The list of the optimizations can by no means considered complete. I have filtered out the low-level optimizations that seem too specific for the particular compilation model. I also am not deeply involved in the declarative language compilation research, so I probably missed quite a few relevant techniques; I would be grateful to anyone pointing my attention to them.

Acknowledgments

This research was supported by SuSE Labs.

References

[W83] D.H.D. Warren, *An abstract Prolog instruction set*, Technical Note 309, SRI International, Menlo Park, CA, October 1983.

[A84] L. Augustsson, *A Compiler for Lazy ML*, Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming, August 1984, pp. 218–227.

[J84] Johnsson, T., *Efficient compilation of lazy evaluation*, Proceedings of the SIGPLAN'84 Symposium on Compiler Construction, June 1984, pp. 58–69.

[FW87] J. Fairbairn, S. Wray, *TIM—a simple lazy abstract machine to execute supercombinators*, Functional Programming Languages and Compiler Architecture, LNCS 274, Springer Verlag.

[W90] P. Wadler, *Deforestation: transforming programs to eliminate trees*, Theoretical Computer Science 73 (1990), 231–248.

[HK92] P.G. Harrison, H. Khoshnevisan, *On the synthesis of function inverses*, Acta Informatica, 29(3):211–239, 1992.

[J92] S.P. Jones, *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine*, Journal of Functional Programming 2(2) (April 1992), pp. 127–202.

[JL92] S.P. Jones, D. Lester, *Implementing functional languages: a tutorial* Published by Prentice Hall, 1992.

[R94] P. van Roy, *Issues in implementing logic languages*, Slides, École de Printemps, Châtillon/Seine, May 1994.

[CL95] M. Cierniak, W. Li, *Unifying Data and Control Transformations for Distributed Shared-Memory Machines*, Proceedings of the ACM SIGPLAN Conference of Programming Design and Implementation (PLDI'95), La Jolla, California, USA, 1995.

[G96] A. Gill, *Cheap deforestation for non-strict functional languages*, PhD thesis, University of Glasgow, Jan 1996.

[P97] G. Puebla, *Advanced Compilation Techniques based on Abstract Interpretation and Program Transformation*, Ph.D. Thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, November 1997.

[L98] Y.A. Liu, *Dependence Analysis for Recursive Data*, Proceedings of the 1998 International Conference on Computer Languages, 1998.

[CAZ99] F. Corbera, R. Asenjo, E.L. Zapata, *New shape analysis techniques for automatic parallelization of C codes*, Proceedings of the 13th International

Conference on Supercomputing, Rhodes, Greece, 1999.

[LS99] Y.A. Liu, S.D. Stoller, *From recursion to iteration: what are the optimizations?*, Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation, pp. 73–82.

[FPP00] F. Fioravanti, A. Pettorossi, M. Proietti, *Rules and Strategies for Contextual Specialization of Constraint Logic Programs*, Electronic Notes in Theoretical Computer Science, Vol. 30 (2) (2000)

[G00] C. Grelck, *Array Padding in the Functional Language SAC*, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2000, June 24–29, 2000, Las Vegas, Nevada, USA

[DC01] D. Diaz, P. Codognet, *Design and Implementation of the GNU Prolog System*, Journal of Functional and Logic Programming (JFLP), Vol. 2001, No. 6, October 2001.

[HS02] F. Henderson, Z. Somogyi, *Compiling Mercury to high-level C code*, Proceedings of the 2002 International Conference on Compiler Construction Grenoble, France, April 2002.

[JM02] S.P. Jones, S. Marlow, *Secrets of the Glasgow Haskell Compiler inliner*, Journal of Functional Programming 12(4), July 2002, pp393-434.

[LA02] C. Lattner, V. Adve, *Automatic Pool Allocation for Disjoint Data Structures*, ACM SIGPLAN Workshop on Memory System Performance (MSP), Berlin, Germany, June 2002.

[LSLR02] Y. Liu, S. Stoller, N. Li, T. Rothamel, *Optimizing Aggregate Array Computations in Loops*, Technical Report TR 02-xxxx, Computer Science Department, State University of New York at Stony Brook, Stony Brook, New York, July 2002.

[H04] J. Hubicka, *The GCC call graph module: a framework for inter-procedural optimization*, accepted to the 2nd GCC & GNU Toolchain Developers' Summit, June 2004.