Overview and Status of DoradoLisp

Richard R. Burton, L. M. Masinter, Daniel G. Bobrow,
Willie Sue Haugeland, Ronald M. Kaplan and B. A. Sheil

Xerox Palo Alto Research Center

Abstract: DoradoLisp is an implementation of the Interlisp programming system on a large personal computer. It has evolved from AltoLisp, an implementation on a less powerful machine. The major goal of the Dorado implementation was to eliminate the performance deficiencies of the previous system. This paper describes the current status of the system and discusses some of the issues that arose during its implementation. Among the techniques that helped us meet our performance goal were transferring much of the kernel software into Lisp, intensive use of performance measurement tools to determine the areas of worst performance, and use of the Interlisp programming environment to allow rapid and widespread improvements to the system code. The paper lists some areas in which performance was critical and offers some observations on how our experience might be useful to other implementations of Interlisp.

I. Background

Interlisp is a dialect of Lisp whose most striking feature is a very extensive set of user facilities including, for example, syntax extension, error correction, and type declarations. It has been in wide use on a variety of time shared machines over the past ten years.

AltoLisp

In 1974, an implementation of Interlisp for the Alto, a small personal computer, was begun at Xerox PARC by Peter Deutsch and Willie Sue Haugeland [Deutsch, 1973]. This AltoLisp implementation introduced the idea of providing a microcoded target language for Lisp compilations which modelled the basic operations of Lisp more closely than a general purpose instruction set. A similar instruction set was also implemented for Maxc, a microprogrammed machine running the TENEX operating system [Fiala, 1978].

The design of AltoLisp is presented in [Deutsch, 1978]. Its characteristics include a very large address space (24 bits); deep binding; CDR encoding [Bobrow & Clark, 1979]; transaction garbage collection [Deutsch & Bobrow, 1976]; and an extensive kernel implemented in a mix of microcode and Bcpl. Although AltoLisp was completed and several large Interlisp programs were run on it, its performance was never satisfactory, due principally to the limited amount of main memory and the lack of support in the processor architecture for either virtual memory management or byte code decoding. DoradoLisp is the result of transferring AltoLisp to an environment with neither of these limitations.

DoradoLisp

The Dorado [Lampson & Pier, 1980] is a large, fast, microcodable personal machine with 16-bit data paths. It has a large main memory (~1 megabyte) and hardware support for both instruction decoding and virtual memory management.

The Dorado had microcode to emulate the Alto, so the initial transfer of the running AltoLisp system to the Dorado was straightforward. Although the microcode to interpret the Lisp instruction set needed to be rewritten, the Bcpl runtime support system was transported with only minor changes. However, initial performance was far worse than would be expected from a simple consideration of machine features. We expected DoradoLisp to dominate Interlisp running on a DEC KA-10, but in fact, some computations took 10 to 100 times longer on the Dorado. The primary goal of the DoradoLisp implementation, then, was to improve the performance of the existing system. First, careful measurements were taken of the system doing a variety of tasks. Functions which took inordinate amounts of time were examined in detail. Additional microcode was written, and major portions of the Lisp code were redone.

The most surprising thing to us was that we obtained considerable performance improvements by moving large parts of the system from Bcpl into Lisp. This allowed us to use a number of programming tools in the Interlisp system, and allowed us to put more structure into the layers of the system's kernel. DoradoLisp is now supporting a user community. While speed ratios vary widely across different classes of computation, it appears that DoradoLisp runs five times faster than a single-user DEC KA-10.

II. The "lispification" of DoradoLisp

Much of the Interlisp system is written in Lisp itself, resting on a kernel not defined in Lisp. The Interlisp virtual machine specification [Moore, 1976] attempted to identify a set of kernel facilities which would support the full Interlisp system. This was done by carefully documenting those parts of the PDP-10 Interlisp system that were written in assembly language or imported from the operating system. This specification is quite large. AltoLisp reduced this kernel by implementing some of the VM facilities in Lisp; DoradoLisp accelerated this development. In addition to improving the transportability of the implementation, the move also improved performance, gave the implementors access to more a more powerful implementation language and programming tools, and limited the breadth of expertise required of system implementors.

Efficiency

Programs written in a higher level language are often less efficient than equivalent assembly language programs, because they cannot exploit known invariances and optimizations which would violate the strict semantics of the target language. Moving code from Lisp into the kernel has been a traditional way of improving the performance of Lisp systems. Substantial sections of the PDP-10 implementation of Interlisp, for example, are in machine code for this reason. When a large proportion of AltoLisp was moved from Bcpl into Lisp in order to improve memory utilization and aid modification, the speed of the system decreased by nearly a factor of three [Deutsch, 1978]. Thus, to improve DoradoLisp performance, we first looked for Lisp-coded sections of the system that could be incorporated into the Bcpl kernel. However, we soon discovered that the poor performance was due more to the design of the algorithms in the kernel than to the language in which they were implemented. Since we did not wish to carry out a large-scale redesign in the limited Bcpl programming environment, we decided to go in the other direction: we would move code *out* of the extended Bcpl kernel and into Lisp so that we would be better able to change the algorithms. Specific targets for replacement were large sections of the Bcpl kernel with known performance problems whose functionality could easily be expressed in Lisp; one of the major areas was the I/O system.

Language power and tools

A primary reason for implementing the bulk of a programming system in itself is that one obtains the advantage of programming in a (presumably) more expressive and powerful language. In addition, we felt that the major modifications and tuning that would be necessary to provide adequate performance would be far more tractable in Interlisp. In Interlisp we had both a first rate programming environment and instrumentation tools, and we had no other system implementation language which had either. Our subsequent experience has sustained this view.

Linguistic uniformity

An important sociological benefit of having a programming system described in the language it implements is that the system's implementors and users share the same culture. Users can inspect the system code, comment on it, adapt it for their own purposes, and sometimes even change it. This involves the users of the system in its design and maintenance in a way that would not be possible if system construction took place in a different language culture. Specifically, the availability of the system source code allows the system to grow and adapt much more rapidly than environments in which a formal documentation phase is a prerequisite to the development and distribution of new facilities. In turn, the users can explore the behavior of the system "all the way to the edges", as there are no sharp language barriers. The value of this linguistic uniformity has been confirmed by its successful use in other language cultures, such as Smalltalk [Goldberg, 1980].

An example: the IO system

A high level language I/O system consists of both low level device handlers and device independent sequential and random access. In most Interlisp implementations, the entire I/O system, up to and including the functions defined in the virtual machine, is provided by the host operating system. In DoradoLisp, all of the logical I/O system and a substantial proportion of the device dependent code is written in Lisp. The logical I/O system implements the Interlisp user program I/O facilities and the underlying operations in terms of which these are implemented. These include sequential and random access operations (i.e., read and write a byte, query end of file, reposition file pointer, etc.), buffer management (both for system only and directly user accessible buffers) and a device independent treatment of file properties. The logical level is in turn implemented in terms of the notion of an I/O *device*. This is an object which provides a standard set of low level, device dependent functions, such as those to read and write a page, create and delete files, etc. Using this interface, the addition of a new device is simply a matter of writing a new set of these functions. The DoradoLisp I/O system design is extensively described in [Kaplan *et al.*, 1980].

III. Implementation techniques

Measurements

In tuning the performance of a program, it is crucial to be able to determine exactly where time is being spent. With a large body of code and limited manpower, it is not possible to "optimize everything." Our performance measurement system has proved invaluable in tracking down specific (and unforeseen) problems.

The measurement system was originally developed for AltoLisp by Deutsch and Haugeland. It operates in two stages. First, the computation of interest is run with *event logging* enabled. This produces a (very large) file of *log events*, which is later analyzed. The log events are put out by both the microcode and the run time support system and include time-stamped events for function call and return, entry and exit from the Bcpl routines, I/O activity, and other events of interest. Alternatively, the microcode can also collect counts of opcode frequencies and a frequency sample of the microcode PC.

Statistics gathering can be enabled at any time that Lisp is running. One can decide spontaneously to take measurements whenever performance unexpectedly degrades. Comparison of these measurements with those taken during a similar run that exhibited normal performance can be used to identify the source of intermittent performance problems. This technique was used, for example, to track down an intermittent slowdown in the code that handled stack frame overflow.

The analysis phase reads the log file and computes summary statistics from it. From call and return events, the time spent in individual functions can be computed, either including or excluding the time spent in the functions called by them. The accumulated times (including the times spent by called functions) locate the higher level functions which are the root of a large amount of time and which may be a candidates for redesign. The individual time (excluding called functions' times) are useful for isolating what improvement can be expected from optimizing or microcoding the body of that function.

Function performance data is presented in tables which show the number of times each function was called and the time spent in each function. For example:

function	#ofCalls	Time	%ofTime	PerCall
NTHCHC	1977	236702	10.6	119
\HT.FIND	1729	168492	7.6	97
LITLEN	2111	131708	5.9	61
LITBASE	2141	118902	5.3	56
...				

Tables such as this isolate very accurately those functions which are worth rewriting as well as identifying those which are not. In this example, NTHCHC, which calls both LITLEN and LITBASE, is an obvious candidate. In another run we discovered that 15 percent of the time was being spent adding one to a counter which had overflowed the small number range. This prompted a redesign of the large number arithmetic.

Additional controls on the analysis routines allow more specific questions to be answered. The analysis can be restricted to that part of the computation within any particular function. For example, only that part of the computation that takes place within READ can be analysed. The analysis can also be limited to a set of functions, in which case only these functions will appear in the table of results. Any time spent in a function not in the set will be charged to the closest bounding function that is.

The analysis routines extract from the log file useful information besides performance data. For example, the dynamic calling behavior is captured in the log, so one frequently useful technique is to list which functions have called (and been called by) other functions, and even how many arguments they were passed. The flexibility of the analysis routines combined with the wealth of information collected during the logging stage allows a given computation to be examined from many points of view.

Initialization

There are several areas that cause fundamental problems for the implementation of a language system in itself: memory management (which requires that the memory manager itself will not cause memory faults), stack overflow recovery (where the stack manager must itself have some stack), and initialization. Initialization is difficult because the initialization program must operate when the system is not in a well formed state. The problem in initialization can be characterized by the question: "If the compiled code reader is itself compiled code, who will read *it* in?"

Several methods of doing initialization suggest themselves. For example, the image can be initialized by a program written in some other language. This is the solution adopted in AltoLisp. Alternatively, the interpreter can be coded in some other language and the compiled code reader can be run interpretively to read itself in. Both of these solutions require a substantial body of non-Lisp code either for storage allocation or for interpretation.

We adopted still another solution. The compiled code reader was modified to load code into an environment other than that in which it is running. The primitive functions that the loader uses to manipulate the environment (e.g., fetch and store into specified virtual memory locations) are replaced by functions that manipulate another memory image stored as a file. To begin with, an empty memory image file is created and then the "indirect" version of the compiled code reader is used to load into this empty image the compiled files that constitute the lowest level of the system. We thus avoid the potential problem of maintaining two different programs which embody knowledge of system data structures.

An appropriate programming environment

One of the strong advantages of writing most of the kernel in Lisp is that Interlisp provides a very powerful programming environment. Some of the tools we found particularly useful are:

Language features: The advantages of "data-less" or data-structure-independent programming have long been known: more readable code, fewer bugs, the ability to change data structures without having to make major source program modifications. The Interlisp record package and data type facility encourages this good practice by providing a uniform and efficient way of creating, accessing and storing data symbolically, i.e., fields of data structures are referred to by name. Because the DoradoLisp implementation allows a large number of data types, we have felt free to give system data structures (such as file-handles, page buffers, read tables) their own data types. In addition, records could be overlaid on structures not under Lisp's control (e.g., the leader page of a disk file or the format of a network packet) to provide the same uniform access.

Cross compilation: We maintained an Interlisp-10 environment in which we could edit, compile and examine functions for the Dorado. The function and record definitions for the Dorado implementation were kept on property lists instead of defintion cells. This allowed us to work on functions such as READ and CONS without destroying the environment in which we were working.

Masterscope: Many of our improvements to AltoLisp involved massive changes throughout the many system source files. Interlisp's Masterscope program was an essential aid in determining what would be affected by a proposed improvement and in actually performing the necessary edits. Masterscope is an interactive program for analyzing and cross-referencing Lisp functions. It constructs a database of which functions call which other functions, where variables are bound, used, or set, and where record declarations are referenced. Masterscope utilizes the information in the database to interpret a variety of English-like commands. Our cross-compilation environment incrementally updated a database that was shared among all programmers on the project, so that with very little overhead the information in the database was kept consistent with the current state of the evolving system.

Masterscope was most helpful in planning and carrying out modifications to major system interfaces, which usually meant changing the numbers and kinds of arguments to various functions. We would first ask Masterscope to simply list the callers of those functions to give some estimate of the impact of the proposed change, much as one might use a static cross-reference program. We would then invoke the SHOW command, instructing Masterscope to locate in the source-file definitions of all the callers the expressions that actually called the interface functions. These expressions were gathered together and displayed as a group, so that we could verify our intuitions about what assumptions clients were making about the interface. In many

cases, the rapid source-code exploration that Masterscope made possible revealed flaws in our redesign which otherwise would not have become apparent until much more effort had been expended. Having decided that our modification was acceptable, we used Masterscope's EDIT command to actually drive the editing. This caused Masterscope to load the definitions of all the client functions, call the Interlisp editor on each one, and position the editor at each of the expressions that needed to be changed. Masterscope, not the programmer, kept track of which functions had been changed and which still needed to be edited. When Masterscope finished the editing sequence, the programmer was sure that the changes had been made completely and consistently.

Our redesign of the I/O system [Kaplan *et al.*, 1980] is a good illustration of the power of this interactive tool. We completely replaced the lowest-level I/O interface, which involved changes to approximately 40 functions on 15 source files. The major part of the revision was accomplished in response to a single EDIT WHERE ANY CALLS '(BIN BOUT ...) command, without ever looking at hard-copy source listings.

Rapid access to system sources: Our cross-compilation environment maintained a shared data base which allows the definition of any Lisp function to be retrieved for viewing or editing in a few seconds. The microcode and Bcpl can be "browsed" using the same interface. Rapid online access to system sources lessened the need for working from listings.

Levelling

One of the original motivations for having a large part of AltoLisp in Bcpl was the belief that it was important not to provide Lisp primitives that gave unrestricted access to the implementation data structures. This reasoning fails to discriminate between the system implementation and user program levels. Allowing system programs arbitrary access to memory locations does not at all imply that user level code has this access.

Failing to make the system/user distinction hurt AltoLisp in three ways. First, it provided one motivation for the large Bcpl kernel. Second, most of that part of the system which *was* written in Lisp was prohibited from manipulating underlying data structures except through overly general functional interfaces. Last, it discouraged the use of higher level structuring facilities (such as the record package) so that code that required any knowledge of system data structures tended to be written entirely in terms of low level primitives.

Using Lisp as a system implementation language requires very careful consideration of the layering of the system into levels of access and knowledge. Further, the precision that is needed cannot be obtained by simple binary discriminations but must be carefully considered for each piece of code. This presents a considerable challenge to the implementors' self restraint, as Lisp provides few facilities to enforce such a layering. Appropriate use of abstraction is essential if layering is to be preserved under the constant revision necessitated by intensive performance debugging.

Diagnostics

Development of the Lisp microcode was aided by a reasonably complete set of microcode diagnostics written in Lisp. Diagnostics are difficult because they are most useful when very little can be assumed *a priori* to work. It is also difficult to achieve complete coverage of all cases. In addition, extensive knowledge of the Lisp system was required to develop diagnostics. For example, every opcode needs to be tested when encountering page faults or stack overflows. Setting up a situation which will page fault or overflow the stack in the next opcode requires a very intimate knowledge of the implementation. Having undertaken several microcode revisions, development of a comprehensive set of diagnostics seems well worth the effort.

Important performance issues

While not strictly a technique, we feel that it is important to mention the major areas in which performance has proved to be crucial. While some of these are undoubtedly specific to DoradoLisp, we feel that they deserve consideration by those who might be building similar Lisp systems.

The earlier intuition that the hardware assist for decoding byte opcodes was important was substantiated. Performance improved by nearly a factor of two when this was installed. Implementing the decoding and dispatch in microcode is conceding a large performance loss.

There are several parts of the system for which it seems important to have microcode support. When written in Lisp, the garbage collector seems to consume between 10-30% of the processor, although the figure varies widely over different computations. Further, in a system that uses deep binding, some form of microcode assist for free variable lookup is very desirable. A speedup factor of between two and four accompanied the introduction of microcode support for this in DoradoLisp. Statistics show that less than one percent of the execution time is now spent in free variable lookup.

Their heavy use in implementing system code almost mandates that the arithmetic functions have complete microcode support. Further, we found it to be critical to have a large range of small numbers (numbers without boxes), so that the performance critical, low level system code did not invoke Lisp's storage management.

IV. Why is an Interlisp implementation so hard?

The Dorado implementation of Interlisp took many times the expected effort to complete. Given the widespread intuition to the contrary, it is perhaps worthwhile to reflect on why it has proved so difficult. The answer is painfully simple: Interlisp is a *very* large software system and large software systems are not easy to construct. DoradoLisp has 17,000 lines of Lisp code, 6,000 lines of Bcpl, and 4,000 lines of microcode. In many ways, the more interesting question is why does it look so straightforward?

Without a doubt, the perceived ease of implementing Interlisp springs from the existence of the virtual machine (VM) specification. This admirable document purports to give a complete description of the facilities that are assumed by the higher level Interlisp software, and does a remarkable job of laying out the foundations of this very large software confederation. It is difficult to resist the implication that a straightforward implementation of this mere 120 pages of specification, much of which is already described in programmatic form, will constitute a new implementation of Interlisp. The issue is rather more complicated than that.

The VM specification looks small, but it is not. There is no simple correspondence between the size of a specification and the volume of code required to implement it. Many of the major problems of an Interlisp implementation (e.g., performance, the garbage collector, the compiler) are simply not addressed at all. We caution Interlisp implementers that the slimness of that document is misleading.

Further, while the virtual machine specification is an excellent first pass, it is far from complete. Many "incidental" functions and variables were left out (e.g. HOSTNAME). It is occasionally ambiguous in places where the system code relies on a specific interpretation. Even though once complete, changes in the higher level code required that the VM be extended to support new facilities. Finding all these variations is an exhausting task. It is substantially easier to get 95% compatibility than 99.9%, and amazing how many programs are sensitive to the difference.

One way to look at the Lisp kernel that was written for DoradoLisp is as the definition of a new VM specification *in Lisp* code. While much of the code is specific to the Dorado environment, a great deal of it simply extends the virtual machine downwards by providing a much lower level treatment of functions such as PRINT and READ. We hope our work will prove useful to others as a firmer foundation for new implementations than that provided by the VM document alone.

Another problem for any very large software system is the existence of a long development tail. A version of DoradoLisp was "sort of running" years ago. Several other implementations of Interlisp have "sort of run" but have never reached production status. One of the key problems here is performance. The success of the PDP-10 implementation of Interlisp is due to a lot of hand tuning. Any obvious clean implementation will prove to be slow, and finding performance problems is difficult, even with good measurement tools. A large number of design decisions have to be made and a large amount of code has to be written. While not all of the decisions have to be optimal, none of them can be pessimal. While the DoradoLisp experience can provide some guidance, many of these decisions will be environment specific.

Finally, an important issue has been compatability with the PDP-10 implementation of Interlisp. In some ways our determination to remain compatible has helped. Ambiguities and omissions from the VM specification could always be resolved by copying the PDP-10 implementation. However, this compatibility requirement was also a burden. Complete compatibility with another implementation is hard. This is particularly so when the new implementation is in a quite different environment (a personal rather than a time-shared machine). The tension between remaining compatible versus exploring the possibilities of a personal machine environment is a continuing issue, which will probably be a focus of our further efforts on the DoradoLisp system.

Acknowledgements

Peter Deutsch was a principal designer and motivating force behind AltoLisp, of which DoradoLisp is a successor. Warren Teitelman has made major contributions to the DoradoLisp project. Martin Kay, Henry Thompson, Richard Fikes and Austin Henderson have also contributed time and effort on various aspects of the project.

References

Bobrow, D. G. & Clark, D. W. Compact encodings of list structure. *ACM Transactions on programming languages and systems* 1, 1979.

Deutsch, L. P. A Lisp machine with very compact programs. *Proceedings of the third international joint conference on artificial intelligence*, Stanford 1973.

Deutsch, L. P. Experience with a microprogrammed Interlisp system. *IEEE Micro-11 Conference*, 1978.

Deutsch, L. P. & Bobrow, D. G. An efficient incremental, automatic garbage colector. *CACM* 19:9, 1976.

Fiala, E. R. The Maxc systems. *IEEE Computer* 11, May 1978.

Goldberg, A. *Smalltalk: Dreams and schemes.* Xerox PARC, to appear.

Kaplan, R. M., Sheil, B. A., & Burton, R.R. The DoradoLisp IO system. Xerox PARC, to appear.

Lampson, B. W. & Pier, K.A. A Processor for a High-Peformance Personal Computer. *7th Int. Symp. on Computer Architecture*, La Baule, France, May 1980.

Masinter, L. M. & Deutsch, L. P. Local Optimization in a Compiler for Stack-based Lisp Machines. Proceedings of the 1980 Lisp Converence, Stanford, 1980.

Moore, J S. *The* Interlisp *virtual machine specification.* Xerox PARC report CSL-76-5, 1976.

Teitelman, W. et al., Interlisp *Reference Manual*, XEROX Parc, 1978.