Direct Function Calls in Lisp

Benjamin Zorn Paul Hilfinger

February 17, 1988

Abstract

This paper compares the implementation of direct function calls in SPUR Lisp with the more traditional indirect call implementations found in Maclisp, Franz Lisp, Spice Lisp, Zetalisp, etc. We examine the performance of direct and indirect function calls on the VAX, MC68020, and SPUR architectures. For the SPUR architecture, single indirection slows applications by 3–4%, and double indirection slows applications by 6–8%. The performance benefits of direct function calls are smaller for the VAX and MC68020 architectures. Implementing direct function calls requires maintaining backpointers from each function to all the functions that call it. The additional memory allocated for backpointers ranges from 10–50% of the memory allocated for functions. Maintaining backpointers increases the time to load functions by 10–15%. Redefining a function in a direct call implementation adds up to 50% to the time needed in an implementation with indirect function calls.

Introduction

This paper discusses different ways of implementing function calls in Lisp. Lisp function calls are interesting because Lisp supports dynamic function definition; any time during the execution of a Lisp program a new function can be defined or an existing function can be redefined. Dynamic function definition contributes to the suitability of Lisp for exploratory programming.

Lisp systems have traditionally facilitated dynamic function definition at the expense of program execution time. When a function is redefined, all functions that call the old definition must be updated to call the new definition. In many Lisp implementations, calls to a function are directed through a location associated with the function's name. When the function is redefined, only a single location needs to be updated. We refer to such function calls as *indirect calls*. Indirection adds overhead to every function call. Zetalisp [Moo85], Interlisp [Xer83], Spice Lisp [WFG85], Maclisp [MAC74], Franz Lisp [FSL*85], and Franz Allegro Common Lisp [Fod87] all implement function calls indirectly.

Direct function calls, in which the destination address of the call is already present in the instruction stream, are the fastest alternative to indirect function calls. Because Lisp functions can be dynamically redefined, direct function calls require modifications to call instructions as functions are redefined. SPUR Lisp is the first Lisp implementation to

This research was funded by DARPA contract number N00039-85-C-0269 as part of the SPUR research project.

implement function calls directly. Some Lisp implementations provide a halfway measure, where functions can be declared local to a particular file and calls within the file are made directly. Such an implementation suffers because once the "local" function calls have been compiled, dynamic redefinitions of the functions are ignored. Our intention was to provide all the capabilities available with indirect calls without the added cost of indirection.

SPUR Lisp is a Common Lisp superset that will run on SPUR, a multiprocessor workstation being designed and implemented at U. C. Berkeley [HLE*85]. SPUR is being designed as a multiprocessor and SPUR Lisp will contain features for multiprocessing, but only the uniprocessor aspects of SPUR Lisp are discussed here. Details about the design and implementation of SPUR Lisp are presented in [ZHHL87]. SPUR Lisp implements function calls directly. Each call instruction contains the address of the actual code of the function being called. Steenkiste estimates that function call indirection in MIPS-X PSL would slow execution by 6% [Ste87]. We have measured the cost of indirection in SPUR Lisp and found it degrades performance from 6–8%.

To facilitate dynamic function definition, SPUR Lisp maintains backpointers from each function to all the functions that call it. We call this set of backpointers the *caller set* for the function. We measured the overhead associated with creating and maintaining caller sets in SPUR Lisp. The size of caller sets varies widely and can reach 50% of the total memory allocated to functions. In addition, building the caller set adds 10–15% to the time needed to load a function. When a function is redefined in SPUR Lisp, all the function's callers must be modified. In the worst case, updating callers can add 50% to the time needed to redefine a function.

Indirect Function Calls

The most common form of function call indirection is *double indirection*. In double indirect implementations, the address of the callee[1] is stored in the *function cell* of the symbol that names the callee. The caller typically stores the symbol naming the callee in the function constants vector. Most Lisp implementations including Zetalisp [Moo85], Maclisp [MAC74], Spice Lisp [WFG85], and SPUR Lisp [ZHHL87] have constants vectors associated with compiled functions.

As an example, consider function `main` which calls function `f`. In a double indirect implementation, the symbol `f` will be stored in the constants vector of function `main`. To call function `f`, the symbol would first be loaded into a register from the constants vector. With the symbol `f` in a register, the function cell of the symbol can be loaded. This cell points to the code that implements the function `f`. Figure 1 illustrates the linkage. Spice Lisp [WFG85], Interlisp [Xer83], Zetalisp [Moo85], and Franz Allegro Common Lisp [Fod87] use double indirect function calls.

A more efficient form of indirection, *single indirection*, only requires one additional load per function call. In single indirection, the *transfer table* is a global table of all compiled function addresses; each function's address is stored at a fixed offset in the transfer table determined at load-time. To call a function, the correct element in the transfer

[1] To avoid confusion, when we discuss function calls, we will refer to the function being called as the *callee* and the function making the call as the *caller*.

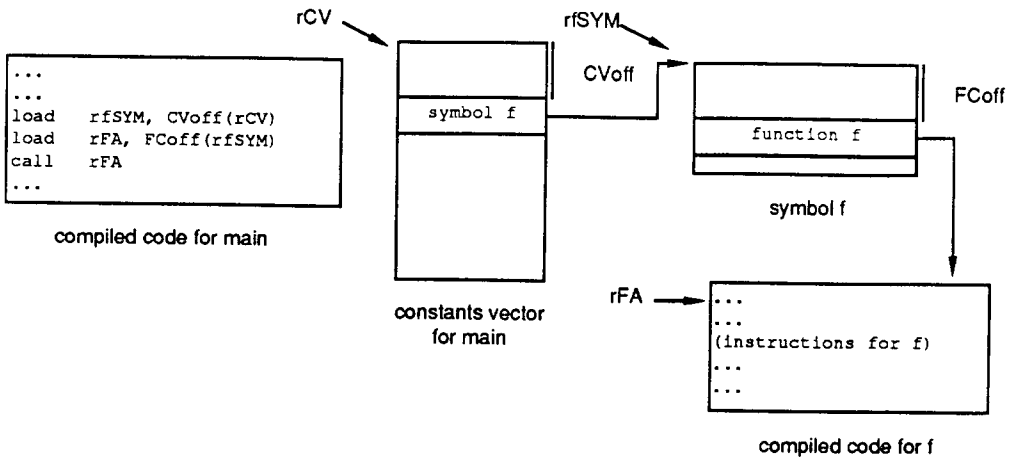

Figure 1: Linkage for Double Indirection. Register names begin with "r". Register `rCV` initially points to the constants vector for the function `main`. `CVoff` is the offset of the symbol `f` in the constants vector for `main`. `FCoff` is the offset of the function cell in a symbol.

table is loaded into a register and the call goes to that address. Figure 2 illustrates this implementation. Due to the possibility of separate compilation, where a function's callees may not be defined in the same file, calls do not initially go through the transfer table. When a function is compiled, the table offset of its callees may not be known. Maclisp handles this problem by making all calls initially go to a linking function. This linking function "unsnaps" the call, making all subsequent calls transfer through the transfer table [MAC74] to the correct address. Maclisp and Franz Lisp [FSL*85] use single indirection.

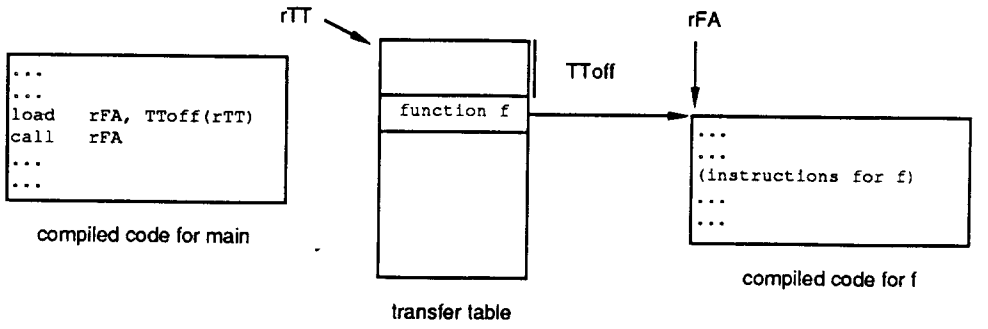

Figure 2: Linkage for Single Indirection. `rTT` is a global register that points to the transfer table. `TToff` is the transfer table offset of the address for `f`.

Direct Function Calls in SPUR Lisp

SPUR Lisp requires two data structures to correctly implement direct function calls: *caller sets*, which identify for each function every function that calls it; and the *unresolved reference list*, a global data structure that records the locations of calls to undefined functions.

Caller sets are implemented in SPUR Lisp as vectors of function addresses (*backpointers*). The caller set for each function is pointed to from the function's constants vector. Each backpointer in a caller set points to a caller of the function. Backpointers do not point to individual call instructions, but to the functions that contain the calls. If a callee has been redefined, the instructions in the caller function must be scanned to find the corresponding call instructions. By only recording the addresses of the caller functions and not individual calling sites, we trade off space used by the caller set for speed of function redefinition.

The unresolved reference list contains locations of calls to undefined functions. When an undefined function is later defined, the loader goes back and fills in the correct value at the appropriate unresolved call instructions. Unresolved references are recorded in a association list. The key in each association is the name of an undefined function and the datum contains information about the locations of unresolved calls. When a function is defined, a check is made to see if there are unresolved references to it.

Execution Overhead of Indirect Function Calls

In this section we estimate the performance degradation associated with indirect function calls. We first identify two parameters we use to estimate the overhead associated with single and double indirection in a generic architecture. We then provide estimates of these parameters for specific Lisp implementations and machine architectures.

Definitions

Function call indirection associates overhead with each function call that could be performed directly. Some calls, such as those performed in a `funcall` or `apply`, require indirection. Measuring SPUR Lisp applications, we found that the fraction of calls requiring indirection is small ($< 10\%$). Based on these measurements, throughout this paper we assume that 90% of the total calls made by a program could be made directly.

In measuring the overhead of indirection, we will examine two quantities. The *direct call fraction* is the fraction of total time an application would spend executing direct function calls if direct function calls were implemented. The direct call fraction only includes the time executing the call instruction (i.e., save a return address and perform a jump), and not time spent passing arguments, setting up frame pointers, etc. In a system with indirection, each call instruction will take longer because of indirection. The *indirection ratio* is the relative cost of an indirect call compared to a direct call, and computed as $T_{indirect}/T_{direct}$, where $T_{indirect}$ and T_{direct} are the unit times for indirect and direct calls, respectively. By computing the direct call fraction and the indirection ratio, we can determine the total execution overhead caused by indirect function calls.

The Indirection Ratio

The indirection ratio is related to the relative costs of calls and loads from memory. A simplistic estimate would be that for single indirection, the indirection ratio can be roughly computed as $IR_{single} = (T_{load} + T_{call})/T_{call}$, where T_{load} is the unit time of a load from memory. Likewise, the double indirection ratio is $IR_{double} = (2 \times T_{load} + T_{call})/T_{call}$. Such simplistic measures are sufficient for load/store architectures that lack indirect addressing modes, such as SPUR and the Am29000 [Joh87]. For architectures with indirect addressing modes, such as the MC68020 [Mot85] and the VAX [DEC81], we have measured the indirection ratio empirically by timing assembly code sequences.

Table 1 presents the indirection ratios for SPUR. In this case, the ratios are computed directly from the instruction sequence. In SPUR, load instructions require two cycles to complete. The instruction after the load cannot use the destination register of the load as an operand. The call instruction also takes two cycles, but in SPUR Lisp the extra cycle is always used to pass the number of arguments. Double indirection requires two loads and a call, and hence 5 cycles. Without code reorganization, the instructions after loads are set to no-ops. With code reorganization [HG83], some of the no-ops can be filled with useful instructions. Hennessy finds that typically 50–60% of the no-ops can be eliminated. The SPUR ratios are computed assuming that 60% of the no-op instructions after the loads can be filled with code reorganizing techniques. The indirection ratio on the Am29000 would be similar to SPUR because both machines have two-cycle load instructions.

SPUR function call	cycles per call (with no-ops)	cycles per call (no-ops filled)	indirection ratio (no-ops filled)
direct	1	1	—
single indirect	3	2.4	2.4
double indirect	5	3.8	3.8

Table 1: Indirection ratio for SPUR function calls. Cycles per call indicates the number of machine cycles necessary to perform only the function call (2 cycles per load plus 1 cycle per call). Cycles per call (with no-ops filled) and indirection ratio are computed assuming that 60% of the no-ops after the loads can be eliminated through code reorganization.

Table 2 presents the indirection ratios for the VAX 8650. The `jsb` VAX instruction was used to perform the call. In measuring the indirection ratio, we inserted `jsb` instructions into a large C program. We then timed the program with no `jsb`'s, direct `jsb`'s, `load+jsb`, etc.

Table 3 presents the indirection ratios for the Motorola MC68020. As for the VAX, the empirical values were computed by timing calls interspersed in a large C program.

The Direct Call Fraction

The direct call fraction depends strongly on the application and the architecture. We present the exact direct call fraction for several SPUR Lisp applications and a probabilistic

VAX function call	nanoseconds per call	indirection ratio
direct call	880	—
single indirect	1140	1.3
double indirect	1410	1.6

Table 2: Indirection ratio for VAX function calls. Nanoseconds per call indicates the time per call computed from averaging 4 data sets of 12,000,000 repetitions on a VAX 8650. Call/return pairs were timed. We determined empirically that for a call/return pair, 50% of the time is spent executing the call and 50% of the time is spent executing the return.

MC68020 function call	nanoseconds per call	indirection ratio
direct call	520	—
single indirect	750	1.4
double indirect	960	1.8

Table 3: Indirection ratio for MC68020 function calls. Times were computed from averaging 10 data sets of 12,000,000 repetitions on a diskless Sun 3/75 with 8 megabytes of real memory. Call/return pairs were timed. We also determined empirically that for a call/return pair, 41% of the time is spent executing the call and 59% of the time is spent executing the return.

measure of the direct call fraction for the same applications in other Lisp systems on other architectures.

The Applications

We have measured the direct call fraction in a group of applications that have been ported to SPUR Lisp. Later in this paper we refer to these same applications when we measure the overhead of supporting direct function calls. The applications are:

`RPG` A composite of the larger Gabriel benchmark programs described in [Gab85]. RPG consists of `puzzle`, `traverse`, `frpoly`, `browse`, `boyer`, and `fft`. The numbers for each benchmark are averaged evenly.

`RSIM` An electronic circuit simulator. The RSIM benchmark involves having RSIM simulate a 10-bit counter for 200 cycles. The RSIM application contains about 2500 lines of Common Lisp.

`OPS5` A routing program implemented in OPS5. The OPS5 benchmark involves having OPS5 partially route a circuit for 200 firings of OPS5 rules. An OPS5 interpreter is implemented in 3500 lines of Common Lisp. OPS5 contains more than 1000 OPS5 rules.

`SLC` The SPUR Lisp Compiler. The SPUR Lisp compiler is implemented in 23,000 lines of Common Lisp.

`UASM` The Perq microcode assembler for Spice Lisp. We measure the microassembler assembling several Perq microcode files. The Perq microcode assembler contains about 5000 lines of Common Lisp.

`PCOM` A Prolog compiler. PCOM translates Warren Abstract Machine (WAM) code to optimized SPUR assembly instructions. PCOM is implemented in 4500 lines of Common Lisp.

`RL` RL is a microcode compiler for signal processing applications. RL is implemented in approximately 5000 lines of Common Lisp.

Unfortunately, due to lack of resources, we have not had time to port all of these applications to SPUR Lisp. In particular, we do not have data for the execution of UASM, PCOM, and RL on SPUR Lisp. Likewise, because the SPUR Lisp compiler is specific to the SPUR Lisp system, we have not spent the time to port the SPUR Lisp compiler to other Common Lisp dialects.

Direct Call Fraction Measurements

Because actual SPUR hardware is still being tested, SPUR Lisp is implemented on an instruction level hardware simulator. The simulator is capable of simulating 60,000 SPUR instructions per VAX CPU second, and simulations of more than 200 million SPUR instructions are routine. With this simulator, we have been able to gather detailed statistics about instruction usage and the exact number of cycles executed in particular functions.

Taylor has compared the overall performance of SPUR with other Lisp machines using this simulator [THL*86].

Because we do not have simulators for the VAX and MC68020 instruction sets, we cannot measure the direct call fraction directly. Instead, we have determined the direct call fraction for various Lisp implementations on these architectures probabilistically. Berkeley Unix (4.3 BSD) provides a system signal that allows processes to be stopped every 10 milliseconds. By sampling the program counter every 10 milliseconds, we measured what fraction of the time various Lisp implementations spend executing call instructions. Table 4 contains measurements of the direct call fraction for VAX LISP, Kyoto Common Lisp, and Franz Allegro Common Lisp, as implemented on the VAX and Sun computers. In every case, we measure Lisp programs compiled for maximum speed and minimum safety. Maximum speed insures that the most frequently called routines (like **cons**) are open-coded, and minimum safety prevents extraneous calls to argument and type-checking procedures.

Steenkiste measures that in MIPS-X PSL, saving the return address and performing the call accounts for 5–6% of the total execution time [Ste87]. Our average of 2.75% is somewhat lower than Steenkiste's measurements, probably because we measure programs in which most common operations like **cons** have been open-coded.

Lisp Implementation	Direct Call Fraction (%)						
	PCOM	RL	RPG	RSIM	UASM	OPS5	avg.
SPUR Lisp (SPUR)	—	—	3.1	2.2	—	2.8	2.7
VAX Lisp (VAX 8800)	1.8	2.3	2.4	5.2	2.4	4.8	3.2
ExCL (VAX 8800)	1.7	1.8	2.0	2.1	1.7	1.7	1.8
KCL (VAX 8800)	2.8	2.6	3.0	3.7	—	2.8	3.0
ExCL (Sun 3/280)	2.3	2.8	2.9	2.9	2.9	2.6	2.7
KCL (Sun 3/280)	4.2	2.5	3.0	3.5	—	2.4	3.1
average	2.6	2.4	2.7	3.3	2.3	2.8	**2.75**

Table 4: Direct call fraction for various Lisp applications running on several Lisp implementations for SPUR, VAX and MC68020 architectures. Measurements for SPUR were made using an instruction level simulator. Measurements for the VAX and MC68020 were made using the interval timer signal provided in Berkeley Unix (4.3 BSD). We assume 90% of calls are implementable as direct calls. Unfilled spaces indicate the application was not ported to the particular implementation.

The Total Overhead of Function Call Indirection

We have measured the indirection ratio and direct call fraction for several architectures. The total overhead of indirection is the additional time spent executing indirect function calls (i.e. $(DCF \times IR) - DCF$, where DCF is the direct call fraction, and IR is the indirection ratio. Table 5 summarizes the predicted total overhead of single and double indirection in for the SPUR, VAX, and MC68020 architectures. While the direct call fraction is similar for the 3 architectures, the indirection ratio is much larger in SPUR. These measurements indicate that direct function calls do not significantly improve performance of Lisp on the

VAX and MC68020 architectures. However, on SPUR and other RISC architectures, where the cost of a load is the same or larger than the cost of a call, direct function calls provide a 6–8% performance improvement.

architecture	direct call fraction (%)	single indirection ratio	double indirection ratio	total single indirection overhead	total double indirection overhead
SPUR	2.7	2.4	3.8	3.8%	7.6%
VAX	2.6	1.3	1.6	0.8%	1.5%
MC68020	2.9	1.4	1.8	1.1%	2.3%

Table 5: Predicted overhead of single and double indirection for various architectures.

The Costs of Direct Function Calls

There are three costs associated with the SPUR Lisp implementation of direct function calls.

- Increased implementation complexity.
- Increased memory utilization.
- Decreased performance of loading and reloading functions.

The increase in implementation complexity is the hardest cost to quantify. We simply measure this cost by counting the lines of Lisp code necessary to implement direct calls. In SPUR Lisp, direct calls are implemented in 3 source files that contain 1261 lines of Common Lisp. By comparison, SPUR Lisp as a whole (not including the compiler) occupies 54 files and contains 31966 lines of Common Lisp. We see that the complexity of direct calls is only a small fraction ($< 4\%$) of a complete Common Lisp implementation.

We will look at the other costs of direct function calls more closely in the following sections. We have measured these costs carefully in SPUR Lisp using the applications described in the previous section. Table 6 presents SPUR's performance for loading and reloading these applications.

Memory Usage of Direct Function Calls

Direct function calls require additional memory for two data structures: the unresolved reference list and the caller sets. Memory allocated to the unresolved reference list is reclaimed quickly. As an application is loaded unresolved references are created. By the time the entire application has been loaded, however, all unresolved references associated with the application should have been resolved. Measurements of our benchmarks show that memory allocated to the unresolved reference list is a small part of the total memory allocated ($< 6\%$) during loading.

application	total functions loaded	functions with callers	load time (sec.)	reload time (sec.)
SLC	1362	529	8.9	16.2
RPG	79	55	0.5	0.5
RSIM	153	67	2.1	2.1
UASM	294	89	1.4	1.9
OPS5	379	262	6.5	6.4

Table 6: Summary of load and reload statistics for 5 SPUR Lisp applications. Measurements are made using a hardware simulator for SPUR. Times are computed assuming SPUR executes five million instructions per second. Calls to user functions are not included in the load times reported—only the actual load operations are measured.

Memory allocated to the caller sets remains throughout the lifetime of a function, and hence is of greater interest than the memory allocated to the unresolved reference list. Memory allocated to a function is divided into 3 data structures: the code vector (instructions), the constants vector, and the caller set. Table 7 shows the amount of space allocated to each of these data structures in our five applications and for the SPUR Lisp system by itself. While the amount of memory allocated to the caller sets varies greatly, typically it remains a small fraction of the total memory allocated to functions.

application	total functions loaded	code vector (bytes)	constants vector (bytes)	caller set (bytes)	caller set overhead (%)
SPUR Lisp system	2083	381	86	29	6.2
SLC	1362	436	92	56	10.6
RPG	79	272	86	430	120.1
RSIM	153	433	86	23	4.4
UASM	294	456	113	197	34.6
OPS5	379	190	85	29	10.5

Table 7: Memory allocated to functions in 5 SPUR Lisp applications. Average sizes of the code vector, constants vector, and caller set are provided.

Loading Performance with Direct Function Calls

Direct function calls require additional overhead as functions are being loaded. This overhead can be divided into two parts. First, when a function is loaded, its instructions must be scanned so that all calls it makes can be recorded in the caller sets of the functions it calls. If the function being called does not yet exist, the unresolved call must be noted in the

unresolved reference list. Second, when a function is defined, all the unresolved references to it must be resolved. Table 8 summarizes the overhead of these two operations for our five applications. Interestingly, the SPUR Lisp Compiler, in which caller sets contributed only moderately to the total memory allocated to functions (10.6%), showed the most overhead at load time (13.7%). The overhead at load time for direct function calls appears to be acceptably small (< 15%).

application	total functions loaded	overhead resolving references (%)	overhead recording callers (%)	total overhead (%)
SLC	1362	3.5	10.2	13.7
RPG	79	1.7	3.5	5.1
RSIM	153	0.5	2.1	2.7
UASM	294	2.1	7.2	9.4
OPS5	379	0.8	1.8	2.7

Table 8: Overhead of direct function calls during initial loading for 5 SPUR Lisp applications. Fraction of total load time not including calls to user functions is given. All measurements reflect loading applications that were not previously defined.

A more significant source of overhead at load time with direct function calls occurs when functions are redefined. When a function is redefined in SPUR Lisp, all functions that call that function must be scanned and the calling instructions must be updated. Note that redefining an application does not always take longer than initially defining it (e.g., look at OPS5 in table 6). This can occur because a redefining function does not incur the same overhead from unresolved references as originally defining a function does. We have measured the overhead of updating callers for our five applications, and the results are provided in table 9.

application	functions redefined	overhead redefining functions (%)
SLC	1277	4.4
RPG	81	13.0
RSIM	154	9.5
UASM	170	57.7
OPS5	330	7.9

Table 9: Overhead of direct function calls during reloading for 5 SPUR Lisp applications. Fraction of total reload time not including calls to user functions is given.

The results of table 9 are hard to interpret. For the largest application (the SPUR Lisp compiler), the overhead of redefining functions was quite small. However, in UASM

function redefinition added more that 50% to the total reload time. More applications are required to determine what the typical redefinition overhead will be.

Conclusions

We have examined the performance overhead of function call indirection and provided detailed measurements of the costs of direct function calls in SPUR Lisp. Here are our conclusions.

- The cost of indirection depends on the architecture, and ranges from 1% for the VAX using single indirection, to almost 8% for double indirection with the SPUR architecture. Measurements show that direct function calls are probably unnecessary on the VAX and MC68020 architectures because call instructions are quite expensive relative to the loads required for indirection. On SPUR and other RISC architectures direct function calls improve performance significantly
- The implementation complexity of direct function calls in SPUR Lisp is small compared to the overall complexity of a Common Lisp implementation.
- The memory required for the unresolved reference list is a small part of the total memory requirement of an application ($< 6\%$), and is only used when the application is initially loaded. The memory allocated to caller sets is a sizable fraction of the total memory allocated to functions (10–50%) and a much smaller fraction of the total memory allocated by an application. Moreover, caller set memory is only referenced when functions are defined and redefined, so may not greatly degrade application performance.
- With a direct function call implementation, initially loading a function takes up to 15% longer than an indirect call implementation. Redefining an application already loaded may take up to 50% longer, but our data is too sparse to generalize.[2]
- Because the implementation is straightforward and the performance benefits are comparable to an effective compiler optimization, we believe that direct function calls are certainly worthwhile to implement for SPUR and other RISC architectures. Because the performance benefits of direct function calls are considerably smaller on the VAX and MC68020 architectures, direct function calls may not be worthwhile implement on these architectures.

References

[DEC81] *VAX Architecture Handbook.* Digital Equipment Corporation, 1981.

[2]Because it is relatively unusual to completely redefine an entire application, we believe that adding 50% to the time to redefine a function is acceptable in general. While applications are being debugged, Lisp programmers tend to modify and redefine only small parts of the application. Since the time required to load a single function is negligible, adding 50% to that time may not be disruptive to the programmer.

[Fod87] John Foderaro. Personal Communication, July 1987.

[FSL*85] John Foderaro, Keith Sklower, Kevin Layer, et al. *Franz Lisp Reference Manual.* Franz Inc., Berkeley, CA, 1985.

[Gab85] Richard P. Gabriel. *Performance Evaluation of Lisp Systems. Computer Systems Series*, MIT Press, Cambridge, Massachusetts, 1985.

[HG83] John Hennessy and Thomas Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[HLE*85] Mark Hill, James Larus, Susan Eggers, George Taylor, et al. SPUR: a VLSI multiprocessor workstation. *IEEE Computer*, 19(11):8–22, November 1985.

[Joh87] Mike Johnson. *Am29000 User's Manual.* Advanced Micro Devices, 1987.

[MAC74] *MACLisp Reference Manual.* April 1974.

[Moo85] David A. Moon. Architecture of the Symbolics 3600. In *Proceedings of the Twelfth Symposium on Computer Architecture*, Boston, Massachusetts, June 1985.

[Mot85] *MC68020 32-bit Microprocessor User's Manual.* Motorola, second edition, 1985. Published by Prentice-Hall.

[Ste87] Peter Steenkiste. *Lisp on a Reduced-Instruction-Set Processor: Characterization and Optimization.* Technical Report CSL-TR-87-324, Computer System Laboratory, Stanford Univ., March 1987. PhD dissertation.

[THL*86] George S. Taylor, Paul N. Hilfinger, James R. Larus, David A. Patterson, and Benjamin G. Zorn. Evaluation of the SPUR Lisp architecture. In *Proceedings of the Thirteenth Symposium on Computer Architecture*, June 1986.

[WFG85] Skef Wholey, Scott Fahlman, and Joseph Ginder. *Revised Internal Design of Spice Lisp.* Technical Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, January 1985.

[Xer83] *Interlisp Reference Manual.* Xerox Corporation, October 1983.

[ZHHL87] Benjamin Zorn, Paul Hilfinger, Kinson Ho, and James Larus. *SPUR Lisp: Design and Implementation.* Technical Report UCB/CSD 87/373, Computer Science Division (EECS), Univ. California, Berkeley, October 1987.