A Multiprocessor Execution Profiler

Christopher Burdorf, John Fitch, Jed Marti, Julian Padget

December 1989
This research was sponsored by the United States Army under Contract No. MDA903-86-C-0059.

This Note contains an offprint of RAND research originally published in a journal or book. The text is reproduced here, with permission of the original publisher.

The RAND Publication Series: The Report is the principal publication documenting and transmitting RAND's major research findings and final research results. The RAND Note reports other outputs of sponsored research for general distribution. Publications of The RAND Corporation do not necessarily reflect the opinions or policies of the sponsors of RAND research.
A Multiprocessor Execution Profiler

Christopher Burdorf, John Fitch, Jed Marti, Julian Padget

December 1989

Prepared for the United States Army
A Multiprocessor Execution Profiler

Christopher Burdorf
The RAND Corporation

John Fitch
University of Bath

Jed Marti
The RAND Corporation

Julian Padget
University of Bath

Abstract

Existing profiling tools generally have crude interfaces, are clumsy to use, and monitor only accumulated CPU time and function calls. After examining these programs, we concluded that they are insufficient aids for profiling a large scale multiprocessing system even if they are adequate for manual analysis of a single processor system. We developed a tool that collects the following information: where CPU time is expended, quality and quantity data passed between functions, how much global data is referenced and modified, and how these characteristics differ among processors on the network. To simplify data inspection the profiler has a mouse-driven graphical interface.

We used the system on a number of single and multiprocessor Lisp programs. The profiler proved its usefulness in performance improvement and problem identification. This paper describes the design of the profiler and gives examples of its utilization.

1 INTRODUCTION

Programmers have used execution time profilers to locate hot-spots and inefficiencies in their code for many years. This has been done in three principal ways:

1. by inserting timing code in the program manually
2. by periodically interrupting the program and finding out which function is executing
3. by automatically incorporating timing or counting code in the program during the compilation process

Alas, the inefficient and error-prone manual process is probably the one most frequently used. Existing profiling tools generally have crude interfaces, are clumsy to use, and monitor only accumulated CPU time and function calls. After examining the avail-

able tools, we concluded that they are insufficient for profiling a large scale multiprocessing system even if they are adequate for manual analysis of a single processor system. In particular, we wanted to know: Where is CPU time expended? What is the quality and quantity of data passed between functions? How much global data is referenced and modified? How do these characteristics differ among processors on the network? To this end we have constructed a new execution profiler for concurrent Lisp programs.

We developed this profiler as a support tool for improving the efficiency of a multiple-processor system based on the Time Warp synchronization scheme [Jefferson 1982] [Jefferson 1985]. We use Time Warp for large-scale discrete event simulations. These simulations utilize object-oriented programming techniques and emulate the actions and interactions of many hundreds of objects. The objects themselves have a high degree of built-in intelligence and in many cases act autonomously without a script. The multiprocessing system is composed of many loosely coupled processors each with its own private memory. In the design and implementation of such a system, there is a strong need to monitor the actual activity of each processor. We must monitor both the data traffic and time by object, in order to identify bottlenecks and inefficient code.

In this paper we describe the profiler's characteristics and capabilities, contrasting them with similar systems. We illustrate how the execution profiler is used in practice, using the Time Warp system as an example. We end by describing some deficiencies of our profiler, along with our plans for experimentation and further development.

2 BACKGROUND

The use of profiling as a debugging and tuning tool is well known, being implemented in hardware on some of the earliest machines. Its use as a means of determining the actual (as distinct from assumed) behavior of a programming system was well expounded by Knuth [1970] in the case of FORTRAN. Profiling as a means of correcting inefficient algorithms has been reported by Fitch [1977] among others. The role of

profiling as a general tool is shown by its inclusion in a number of commonly used systems. In this section we review the use of profiling in some of these systems, and consider their capabilities against our particular needs.

The widely used UNIX system, in its 4.3 BSD form, includes an advanced profiling system, GPROF [Graham 1982]. GPROF generates execution times and call counts for functions and subroutines, and displays the time spent in the functions and their offspring by the call graph. The information is presented as tables along with explanatory information to aid in reading the tables. This kind of profile allows detailed analysis, but it requires a full monitor file from the run. It is not suited to giving dynamic views of the execution, required for viewing the transitory behavior of the multiprocessor. Also, GPROF is not language specific, so it can't tell us much about LISP. Some of the ideas of GPROF are incorporated within the system described in this paper, but GPROF does not provide a sufficient basis on which to build.

Many Lisp systems provide some profiling capabilities. Cambridge Lisp [Fitch 1977b] can keep counts for each function or code segment. The counts can then be inspected or zeroed at will. This does provide some of the dynamic performance monitoring we require, but the interpretation of the information is difficult, and requires knowledge of the compiled code. The Franz Lisp [Federaro 1983] profiler generates a report listing all the functions profiled, the functions that called them, the functions they called, and the amount of time expended in each. It also lists the total number of function calls during the run, and how those calls were distributed by function. This facility is close to what we want, but again it is a static and separate system, with the information presented in a tabular fashion. The Lisp/VM [Alberga 1986] profiler provides several degrees of profiling, but comes closest to what we need in its most detailed option, which in addition to monitoring function call counts also maintains frequency counts of how often execution was found to be within a particular function when interrupted (in fact, because of the IBM architecture this must be done by polling at function entry and on backward branches).

Melenk and Neun [1986] reporting on the results of using a profiler on Lisp programs on the Cray X-MP, discovered improvements which led to a 40% reduction in CPU utilization by identifying instruction sequences with high memory traffic. Several mechanisms were used. First, the Lisp compiler was modified so that instructions that call functions or access global variables cause the compiler to generate extra counting instructions. This information is collected at the end of the run. The system also has the facility to recognize the calling function and maintain separate counts for each one (the target function is "wrapped" with code to examine the return address of the caller and convert it into a function name). Each function that calls the target function has an associated table entry in which the number of calls it makes is kept.

A third facility, the "SPY" option, runs as a separate process, frequently interrupting the profiled program and accumulating counts of each function in which the program counter is detected. This profiler was used with great success to improve the efficiency of compiled Lisp programs for specific hardware. Unfortunately, only parts of the system are portable. Melenk and Neun's techniques do not address the question of locality of reference in memory traffic, which is more significant on a vector processor.

The Symbolics Lisp machine [Symbolics 1984] does not have a proper profiling mechanism in the sense of the other systems described here. It does provide access to program counter metering, and so allows the determination of the amount of execution time spent in a particular section of code. This capability, together with the advise facility, provides the raw building blocks for constructing a profiler. However, this technique is not portable to other systems.

While all the aforementioned systems provide useful facilities in the environment in which they were conceived, they are insufficient for our analysis requirements. For example, because information transfer is a critical issue in distributed memory systems, we need to analyze the quality and quantity of the data structures passed to functions. For the same reason we also need information about the breakdown of interprocessor function calls. We decided that the simple report format is not sufficient due to the large amount of collected data. A graphical interface is necessary to help us put such large volumes of data into perspective.

3 PROFILER DESIGN

Our execution profiler has many features in common with existing systems, but it differs in a few important aspects. First and foremost, because we are working in a multi-processor environment, information is collected from more than one processor. Second, the information is available both during and after execution rather than requiring off-line analysis. Third, the profiler monitors data types and size as well as CPU utilization. Fourth, information is presented graphically.

We constructed the profiler to help us improve the performance of the message-passing Time Warp system. Consequently, we need to monitor the executing program on each processor and view the data from the perspective of both the entire system and that of a single processor. The profiler is designed as a single module loaded on all processors involved in a program execution. The user's console operates the controlling processor for the run and acts as a central repository for profile information. To minimize impact on the communication network, the profiler collects statistics at the completion of the job or at regularly scheduled intervals. Data passed to the monitoring processor goes through a different mechanism from that through which the data being monitored
passes. Thus, there is no possibility of the monitoring system including itself in the profile. The profiler is not dependent upon use of a particular number or organization of processors. The only unavoidable drawback is that the high overhead may have an impact on multiprocessor performance far in excess of that caused for each processor.

The information collected by the profiler is available for either immediate or off-line analysis. Using the immediate mode, the executing program can be stopped at any point and the profile examined. The program can then be restarted for further profiling. The collected data can be written to disk enabling an off-line analysis either using the graphics interface or some other method. We have found this to be particularly useful for detailed analyses involving relationships between program parameters.

By its nature, Lisp programming involves the creation of data structures of arbitrary shape and size. The run time of a function frequently depends on the data created during its execution. To understand a function's operation in more detail, we monitor the amount and type of information entering the function (via its arguments and from accesses to free variables and property lists) and leaving (via its value and assignments to free variables and property lists). We also monitor the creation of dynamic data structures. This information has frequently pinpointed the cause of an inefficiency as well as providing data for estimating performance of particular Lisp functions.

We opted for a graphical display and mouse-controlled browser because of the number and diversity of functions that must be monitored in the large programs we profile. The display format defaults to a normalized bar chart showing the frequency of system function calls, since this is the most commonly desired information. The mouse can be used to scroll through the data or show selected alternative representations or fields. The browser limits data display in order to avoid swamping the user, although he is free to increase this as he sees fit—it is a question of absorption of information versus the amount of room available in which to display it.

In the sections below, we describe the two schemes by which the profiler collects information. *Invasive* data collection wraps functions with code to monitor their performance. The functions themselves may be reconstructed to perform those operations. *Interrupt driven* data collection is accomplished by periodically interrupting program execution and determining from the address of the program counter which function (or object) was executing.

## 4 INVASIVE DATA COLLECTION

Under this regime we collect function call, data type, and quantity information by wrapping the target function during the loading process. The wrapper examines argument lists and sets local variables for monitoring the creation of dynamic data. When compiled code is loaded, before each function is defined, it is wrapped in a shell of monitoring code. Interpreted definitions are subject to a code-walker which identifies the free variable references and assignments and wraps each in some more accounting code.

Because we are also interested in the quality and quantity of data passed between functions, we monitor the frequencies of the following datatypes and values:

1. small integers
2. large integers (bignums) and the number of digits in them
3. floating point numbers
4. strings and the number of characters in them
5. identifiers
6. NIL
7. code pointers (function addresses)
8. vectors, and their size
9. lists and their length

A function within the invasive data collection code examines Lisp data items and visits structures to their lowest levels counting elements as it goes. This information is kept in various places for the duration of the program's execution.

As stated earlier, when the function is defined it is wrapped in a shell and if it is to be interpreted the body is examined and further data collection code is added. This differs from the Melenk and Neun approach [Melenk 1986] of redefining the compiler to add the appropriate code. Our method has the advantage of being simpler and more portable; theirs has the advantage of efficiency.

The wrapper performs the following actions:

- **Entry**
  1. Build an input data structure (if not extant) to hold function entry statistics.
  2. Examine and measure (as above) each function parameter.
  3. Increment the call count for this function.
  4. Initialize the dynamic data structure counters. These count the number of CONS's, new vectors, and new vector elements created during the execution of this function.
  5. Evaluate the function.

- **Exit**
  1. Build an output data structure (if not extant) to hold function exit statistics. This structure is only created and used when the structure exits normally. Exits through the THROW and ERROR mechanisms do not invoke the wrapper.
  2. Examine and measure the size of the data structure returned.
3. Account for dynamic storage consumption during the execution of function.

4. Store the information on the property list of the function.

All functions are wrapped in the above shell. In addition, interpreted functions are scanned before definition, with code inserted for the following purposes:

1. If a global variable is being assigned a value, the assignment is wrapped with a shell that counts the assignment as well as examining and counting the data being assigned.

2. If a global variable is being accessed, it is wrapped in a shell that counts the accesses and examines and counts the data structure the global contains.

3. All calls to functions except those explicitly flagged not to be wrapped are embedded in code to increment the call count. This element of the system tells you how many times each function called function 2.

To monitor the use of dynamic storage, system allocation functions are redefined to record the size and number of structures they create.

5 AN EXAMPLE OF INVASIVE COLLECTION

The two code fragments shown in figure 1 demonstrate the effects of wrapping the recursive factorial function with all the invasive profiling code. In the wrapped version, the code between lines 1 and 10 is the profile prolog. It walks the actual parameters counting bytes and data types and clears the dynamic storage counters. The code between lines 11 and 18 is the original function definition. However, each function call is wrapped with a wrapper. Examining the counter values will reveal the patterns of function invocation. Lines 19 through 23 catalog the returned values and summarize the number of dynamic items allocated. We ran the wrapped version of the factorial function to calculate the factorials from 1 to 20. The profile showed that over 65% of the calls to Fact return big numbers. Also the profiler showed that there were 32,262 cons cells generated when Fact was run interpreted compared with 12 when run compiled.

6 INTERRUPT DRIVEN DATA COLLECTION

The interrupt driven mechanism operates by repeatedly sampling the program counter and identifying the active function. Assuming that the program is not synchronized with the clock frequency, this information gives an accurate view of which functions are utilizing the most CPU time. We adopt this method rather than timing each function with the system clock for two reasons. First, the UNIX system clock is accurate to only 1/60th of a second. Most small functions take considerably less than this. Second, the timing method gives elapsed time for the whole function rather than time actually spent in the code itself.

The sampling routine is driven off the system clock on an interrupt basis. Before the program is executed, a table of function names and addresses is sorted into ascending order by address. At the start of program execution, the clock is initialized to interrupt every 1/60th of a second. The interrupt service routine samples the program counter and a binary search routine identifies which function is being executed (this takes at most 13 tests of the table). This action occurs asynchronously on every processor in the network. At program completion, the interrupts are disabled and the profiler retrieves the frequency table from each remote machine.
The invasive and interrupt driven techniques can be used simultaneously. However, the results are necessarily perturbed because of the high overhead of the invasive counting mechanism. In general, either one or the other is enabled during a run.

7 DATA DISPLAY

The primary form for the presentation of data is as a bar chart. The chart first gives the total number of calls of each function, sorted in descending order, with the largest bar scaled to fill the window. This initial situation is shown in Figure 2. In general, the number of bars will be greater than can be fitted in a window, so the profiler allows the user to scroll forwards or backwards, either by one line, or by half or a full screen height. The scroll controls can be seen in Figure 2 on the right of the main profiler window (the one in the top left of the screen). As scrolling occurs, the data is re-normalized to take advantage of the full width of the window.

To keep profiler results over long periods of time, we implemented hard copy table output. Some of this is seen in tables later in the paper.

As we stressed in the earlier sections, one of the important aspects of a profiler is the variety of ways in which the same data can be viewed. Accordingly the display part of the profiler can be set to show the number of entries to functions, or the number of exits (which may not be the same because of non-local exits or errors), the number of CONS operations executed within the function, the sizes of the vectors used or created in the function, the number of accesses to non-local data items, or the number of bytes passed to the function as argument structures. It might be considered an oversight that the profiler draws no distinction between data passed to the function from another function and data passed to the function by a recursive call (either direct or indirect). Thus, data may be counted more than once in the case of recursive functions.

Another aspect of the display system is that it is possible to sort the data in many different ways. The default is number of calls, but one can choose number of exits, size of data inputs, length of strings, or any other of the many features measured.

The top level display of the browser shows aggregate information. This information can be broken down and viewed by pointing at any bar and selecting from a number of possible decompositions. For example, the user can examine the number of calls per processor rather than the aggregate total. This information is displayed in a second bar chart such as that shown to the right of the main chart in Figure 2. Similarly, the user can investigate the quality of data for a particular function.

Though the user interface displays only a fixed set of information, the basic graphics routines are available to the programmer. This provides a uniform user interface for data collected from other sources.

![Figure 2: Profile of Time Warp with Functions Wrapped.](image)

For example, we have used this facility to monitor CPU utilization by object rather than function. In this case, the profile data is generated by the run time task scheduler inside the concurrent execution mechanism. There are opportunities here for a fully user-programmable profiling browser in the style of spreadsheets, but we leave this for future work.

8 USE OF THE PROFILER IN A CONCURRENT PROCESSING ENVIRONMENT

We originally constructed the profiler to study multiprocessor LISP programs, and we found it effective for understanding program data flow and for improving programming efficiency. For example, we discovered that a routine that was supposed to be idle-waiting for an input message was in fact busy-waiting. The number of function calls was large and the amount of dynamic storage used was immense, causing frequent (and until then) unexplained garbage collections. Recognizing this error would have been difficult without this tool.

As part of our optimization effort, we have run experiments that exercise parts of the concurrent processing environment with typical data. For example,
8.1 Implementation

The profiler is implemented in the Portable Standard Lisp (PSL) dialect [Kessler 1987]. With the exception of the interrupt service code, we believe it is portable to any system running PSL. The graphics interface is implemented using the X windows version 10.4 [Gettys 1986] through a foreign function interface to the X windows library (written in C). The graphics portion of the program is portable to any PSL system that supports X windows (currently the SUN 3 and HP 300 series UNIX machines).

8.2 Profiler Performance

The profiler affects multiprocessing performance in several ways, even beyond the expected degradation in performance from extra wrappers and interrupts. This extra overhead can greatly affect run time by imbalancing the computation load across processors. Also, the storage overhead for the invasive method can be significant, leading to a marked increase in the number of garbage collections, further slowing execution. To understand these effects, we ran several experiments which we summarize in Table 1. For each experiment, there are three numbers: first, the total run time; second, the execution ratio, the ratio of profiled execution time to unprofiled execution time; and third, the storage ratio, a ratio of profiled code storage consumption to unprofiled code storage consumption.

The test programs differ somewhat in their characteristics. Program A is a data flow analyzer working in a single processor on a small Lisp program [Marti 1983]. The test is run on a SUN 3/260 processor with 16 megabytes of storage. Program B is a concurrent queuing simulation with two active objects. This is run first on two identical SUN 3/160 processors, and then on three processors of differing power (2 SUN 3/160's and a SUN 3/50).

As expected, the invasive collection mechanism has an extremely high overhead. This mainly reflects extra CPU time required to count input and output data structures. Program A operates on small data structures with many calls to unprofiled system functions, while Program B worked on many long data structures (vectors and lists) with many more calls to profiled functions. As can be seen in the table, invasive profiling results in much less overhead with small data structures. Also, the surprising decrease in performance when switching from two to three processors with invasive collection was found to be caused by the difference in processor power between the three machines. The invasive collection overhead is much greater on the smaller, slower processor. This in turn affects the load balance between processors, further slowing the collection process.

9 Example Applications

9.1 The Time Warp system.

The Time Warp system is a distributed simulation environment that attempts to speed up the execution time of large object-oriented simulations (e.g., ROSS [McArthur 1981], Time Warp [Jefferson 1985], RISE [Marti 1985]). Because of the scale of these simulations, it is essential that the Time Warp system be as efficient as possible. Accordingly, we used the profiler to discover bottlenecks and to modify and improve the system.

In the Time Warp system, the current state of the object is saved after each update. The action requires copying the values of the instance variables of the object, and can be very storage intensive. However, we found from a frequency count that the majority of time was spent in a function called COMPRESS, which is used in the object system each time a method is invoked. Our object system, TWBERT, laboriously

<table>
<thead>
<tr>
<th>Group</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>TWBERT</td>
<td>38683</td>
</tr>
<tr>
<td>NETWORK</td>
<td>13152</td>
</tr>
<tr>
<td>LIST PROCESSING</td>
<td>8887</td>
</tr>
<tr>
<td>TIME WARP</td>
<td>6467</td>
</tr>
<tr>
<td>FLOAT</td>
<td>7453</td>
</tr>
<tr>
<td>MISCELLANEOUS</td>
<td>38981</td>
</tr>
<tr>
<td>SYSTEM</td>
<td>71783</td>
</tr>
<tr>
<td>TOTALS</td>
<td>185386</td>
</tr>
</tbody>
</table>
constructs the method name each time a message is sent. Table 2 is a summary of a profile run in which we classified CPU times by type of activity. The Table 2 data shows that the subordinate functions of COMPRESS occupy most of the processing cycle. The system functions occupy 50% of the time. They constitute low level operating system services (eg, I/O, Networking) and low level LISP routines. We plan to allow the frequency count option of the profiler to dig deeper into the LISP system to determine lower level function counts. The miscellaneous functions are ones that were profiled, but which did not fit into any of the specific groups.

The results show that TWBERT takes up approximately 27% of cpu time in a Time Warp execution. Table 3 shows the breakdown on a per processor basis. As can be seen most of TWBERT’s cpu time is spent in the EXPLORER and IMPLODE functions.

Table 3: Profile of the TWBERT Module

<table>
<thead>
<tr>
<th>Function</th>
<th>GNU</th>
<th>SNOWBIRD</th>
<th>ALTM</th>
<th>LEMMA</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREDINET</td>
<td>314</td>
<td>2340</td>
<td>2240</td>
<td>1</td>
<td>7729</td>
</tr>
<tr>
<td>CRREAD</td>
<td>210</td>
<td>2221</td>
<td>2116</td>
<td>0</td>
<td>7597</td>
</tr>
<tr>
<td>CRPRINTD</td>
<td>263</td>
<td>2184</td>
<td>2062</td>
<td>0</td>
<td>7884</td>
</tr>
<tr>
<td>LISPCHAR</td>
<td>227</td>
<td>1876</td>
<td>1722</td>
<td>2</td>
<td>5870</td>
</tr>
<tr>
<td>COMPRESS</td>
<td>220</td>
<td>1710</td>
<td>1616</td>
<td>0</td>
<td>5536</td>
</tr>
<tr>
<td>EXPLORER</td>
<td>188</td>
<td>1132</td>
<td>1075</td>
<td>0</td>
<td>3794</td>
</tr>
<tr>
<td>EXPLORER</td>
<td>231</td>
<td>162</td>
<td>158</td>
<td>0</td>
<td>552</td>
</tr>
<tr>
<td>INTERN</td>
<td>40</td>
<td>46</td>
<td>42</td>
<td>0</td>
<td>132</td>
</tr>
<tr>
<td>IMPLODE</td>
<td>45</td>
<td>50</td>
<td>28</td>
<td>0</td>
<td>123</td>
</tr>
<tr>
<td>GETI</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>GETI</td>
<td>220</td>
<td>145</td>
<td>141</td>
<td>0</td>
<td>506</td>
</tr>
<tr>
<td>TOTALS</td>
<td>18710</td>
<td>21733</td>
<td>21207</td>
<td>3</td>
<td>38683</td>
</tr>
</tbody>
</table>

To get this data, we profiled only the TWBERT module. Table 3 also shows the load per machine. You can see that GNU is heavily loaded while Snowbird is less burdened.

9.2 Profiling The ERNIE Object System

We then developed the ERNIE object system [Florman 1988] to replace the slow TWBERT module. The simulation we use for testing involved a platoon of tanks moving across a road network in West Germany. This simulation is a subset of a larger simulation environment that we are now exploring using ERNIE and Time Warp.

Figure 3 displays the profiler frequency count for the simulation. INITCODE appears at the top as having the highest count, and refers to an interrupt generated in either the operating system or a low level LISP routine that is unidentifiable to the profiler. The remaining functions listed are more specific. These functions such as COERCEDURE, LESSP, and FLOATLESSP are all used in floating point calculations and make up approximately half of the interrupt counts during the run.

The profiler data tells us that the simulation is floating point intensive; since the floating point system now greatly overshadows object system time, it suggests that our LISP system (PSL) is quite slow at generic arithmetic. The result of these improvements is that ERNIE runs roughly three to ten times faster than TWBERT.

10 IMPROVEMENTS

We have identified several areas for improvement in the profiler. As seen from the performance data, more attention must be paid to efficiency, particularly for the invasive approach. Much of the cost can be attributed to measuring data structure sizes. The slowdown from "called by", entrance, and exit counts is
relatively minor.

Recently a facility for converting the display into hard-copy charts and graphs has been added. This is particularly useful for documenting changes in the profiled code over time. More than once we discovered that a supposed "improvement" actually made the code run slower. For example, the addition of the priority request protocol [Gates 1988] to the basic Time Warp model, which was expected to show a significant speedup by capitalizing on side-effect free messages, actually proved to be slower than the indirect request mechanism [Sowizral 1986].

After several months experience of using the profiler, we believe it would also be useful for it to be user extensible. This would be a major task, but it would allow the programmer to monitor performance at a higher level and access the profiler user interface and data. We also feel that the user interface would benefit from a data-base system approach that would supplement the extensibility of the display system by permitting multiple data views.

CONCLUSIONS

The multiprocessing profiler proved useful to a large concurrent processing project by identifying program bottlenecks and program bugs. The multiprocessing profiler graphical interface simplified our inspection of profile generated data. We take advantage of the workstation window system to examine the data from different perspectives. In addition to the usual interrupt driven profiling mechanism, we obtain quantity and quality data by wrapping each Lisp function with profiling code appropriate to the function type.

The profiler and wrapping mechanisms were particularly easy to build in a Lisp system. Applying this methodology to other programming languages would require considerable modifications to their compilers and run time systems. The wrapping code is not suitable to a statically typed language such as C or PASCAL as dynamic type information is not available at run time.

Our use has identified a number of problems to which we propose solutions. The wrapping code impacts system performance to such a degree that its frequent use is infeasible. This becomes problematic in multiprocessor systems where timing can be critical. A more intelligent approach to wrapping recursive functions would eliminate some of the system overhead. However the most reasonable approach is to have the user wrap only the most critical functions determined by the interrupt mechanism.

The information garnered by the profiler is best treated as a data base. A more intelligent approach to this data would allow complex queries and a more general purpose display. An object-oriented design in the display code would have simplified the coding process.

References


