International Journal of Engineering and Advanced Technology (IJEAT)
ISSN: 2249-8958 (Online), Volume-2 Issue-1, October 2012

Lazy Expression Evaluation with Demand Paging In Virtual Memory Management

Y.Soumya,T. Ragunathan

Abstract— In computer operating systems, demand paging (as opposed to anticipatory paging) is a method of virtual memory management. Mainly this paper focus on the process of execution of pages in physical memory. Mainly this paper tells that faulits of page. Mainly these paper exaplin the lazy loading technique. This lazy loading technique performs the evaluation of expressions in the virtual memory management. This paper attempts the Short-circuit evaluation

KeyTerms: demand paging, virtual memory management, lazy loading technique, page fault, Short-circuit evaluation

I. INTRODUCTION:

In computer operating systems, demand paging (as opposed to anticipatory paging) is a method of virtual memory management. In a system that uses demand paging, the operating system copies a disk page into physical memory only if an attempt is made to access it (*i.e.*, if a page fault occurs). It follows that a process begins execution with none of its pages in physical memory, and many page faults will occur until most of a process's working set of pages is located in physical memory. This is an example of lazy loading techniques.

II. BASIC CONCEPT

Demand paging follows that pages should only be brought into memory if the executing process demands them. This is often referred to as lazy evaluation as only those pages demanded by the process are swapped from secondary storage to main memory. Contrast this to pure swapping, where all memory for a process is swapped from secondary storage to main memory during the process startup.

Commonly, to achieve this process a page table implementation is used. The page table maps logical memory to physical memory. The page table uses a bitwise operator to mark if a page is valid or invalid. A valid page is one that currently resides in main memory. An invalid page is one that currently resides in secondary memory. When a process tries to access a page, the following steps are generally followed:

Manuscript published on 30 October 2012.
***** Correspondence Author (s)
Y.Soumya, Associate Professor, Dept of CSE St.Mary's College of Engineering & Technology, Deshmukhi, Hyderabad
T. Ragunathan, Honorary Director International School for Computer Science and Information Technology, JNIAS.

- Attempt to access page.
- If page is valid (in memory) then continue processing instruction as normal.
- If page is invalid then a page-fault trap occurs.
- Check if the memory reference is a valid reference to a location on secondary memory. If not, the process is terminated (illegal memory access). Otherwise, we have to page in the required page.
- Schedule disk operation to read the desired page into main memory.

2.1. Advantages

Demand paging, as opposed to loading all pages immediately:

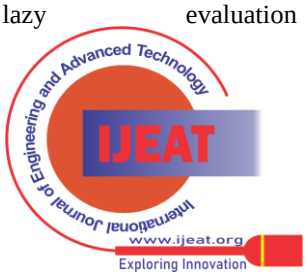
- Only loads pages that are demanded by the executing process.
- As there is more space in main memory, more processes can be loaded reducing context switching time which utilizes large amounts of resources.
- Less loading latency occurs at program startup, as less information is accessed from secondary storage and less information is brought into main memory.
- As main memory is expensive compare to secondary memory, this technique helps significantly reduce the bill of material (BOM) cost in smart phones for example. Symbian OS had this feature.

2.2. Disadvantages

- Individual programs face extra latency when they access a page for the first time.
- Programs running on low-cost, low-power embedded systems may not have a memory management unit that supports page replacement.
- Memory management with page replacement algorithms becomes slightly more complex.
- Possible security risks, including vulnerability to timing attacks; see Percival 2005 Cache Missing for Fun and Profit (specifically the virtual memory attack in section 2).

III. LAZY EVALUATION

In programming language theory, lazy evaluation or call-by-need is an evaluation strategy which delays the evaluation of an expression until its value is needed (non-strict evaluation) and which also avoids repeated evaluations (sharing). The sharing can reduce the running time of certain functions by an exponential factor over other non-strict evaluation strategies, such as call-by-name.[of lazy evaluation include:



Published By:
Blue Eyes Intelligence Engineering and Sciences Publication (BEIESP)

- Performance increases by avoiding needless calculations, and error conditions in evaluating compound expressions
- The ability to construct potentially infinite data structures
- The ability to define control flow (structures) as abstractions instead of primitives

Lazy evaluation can lead to reduction in memory footprint, since values are created when needed. However, with lazy evaluation, it is difficult to combine with imperative features such as exception handling and input/output, because the order of operations becomes indeterminate. Lazy evaluation can introduce space leaks.

The opposite of lazy actions is eager evaluation, sometimes known as strict evaluation. Eager evaluation is commonly believed as the default behavior used in programming languages.

3.1. History

Lazy evaluation was introduced for the lambda calculus by (Wadsworth 1971) and for programming languages independently by (Henderson & Morris 1976) and (Friedman & Wise 1976).

3.2. Applications

Delayed evaluation is used particularly in functional programming languages. When using delayed evaluation, an expression is not evaluated as soon as it gets bound to a variable, but when the evaluator is forced to produce the expression's value. That is, a statement such as x:=expression; (i.e. the assignment of the result of an expression to a variable) clearly calls for the expression to be evaluated and the result placed in x, but what actually is in x is irrelevant until there is a need for its value via a reference to x in some later expression whose evaluation could itself be deferred, though eventually the rapidly growing tree of dependencies would be pruned to produce some symbol rather than another for the outside world to see.

Some programming languages delay evaluation of expressions by default, and some others provide functions or special syntax to delay evaluation. In Miranda and Haskell, evaluation of function arguments is delayed by default. In many other languages, evaluation can be delayed by explicitly suspending the computation using special syntax (as with Scheme's "delay" and "force" and OCaml's "lazy" and "Lazy.force") or, more generally, by wrapping the expression in a thunk. The object representing such an explicitly delayed evaluation is called a future or promise. Perl 6 uses lazy evaluation of lists, so one can assign infinite lists to variables and use them as arguments to functions, but unlike Haskell and Miranda, Perl 6 doesn't use lazy evaluation of arithmetic operators and functions by default.

Delayed evaluation has the advantage of being able to create calculable infinite lists without infinite loops or size matters interfering in computation. For example, one could create a function that creates an infinite

list (often called a *stream*) of Fibonacci numbers. The calculation of the *n*-th Fibonacci number would be merely the extraction of that element from the infinite list, forcing the evaluation of only the first n members of the list.

For example, in Haskell, the list of all Fibonacci numbers can be written as:

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

In Haskell syntax, ":" prepends an element to a list, tail returns a list without its first element, and zipWith uses a specified function (in this case addition) to combine corresponding elements of two lists to produce a third.

Provided the programmer is careful, only the values that are required to produce a particular result are evaluated. However, certain calculations may result in the program attempting to evaluate an infinite number of elements; for example, requesting the length of the list or trying to sum the elements of the list with a fold operation would result in the program either failing to terminate or running out of memory.

In most eager languages, *if* statements evaluate in a lazy fashion.

```
if a then b else c
```

evaluates (a), then if and only if (a) evaluates to true does it evaluate (b), otherwise it evaluates (c). That is, either (b) or (c) will not be evaluated. Conversely, in an eager language the expected behavior is that

```
define f(x,y) = 2*x
set k = f(e,5)
```

will still evaluate (e) and (f) when computing (k). However, user-defined control structures depend on exact syntax, so for example

```
define g(a,b,c) = if a then b else c
l = g(h,i,j)
```

(i) and (j) would both be evaluated in an eager language. While in

```
l' = if h then i else j
```

(i) or (j) would be evaluated, but never both.

Lazy evaluation allows control structures to be defined normally, and not as primitives or compile-time techniques. If (i) or (j) have side effects or introduce run time errors, the subtle differences between (l) and (l') can be complex. As most programming languages are Turing-complete, it is possible to introduce user-defined lazy control structures in eager languages as functions, though they may depart from the language's syntax for eager evaluation: Often the involved code bodies (like (i) and (j)) need to be wrapped in a function value, so that they are executed only when called.

Short-circuit evaluation of Boolean control structures is sometimes called *lazy*.

IV. SHORT-CIRCUIT EVALUATION

Short-circuit evaluation, minimal evaluation, or McCarthy evaluation denotes the semantics of some Boolean operators in some programming languages in which the second argument is only executed or evaluated if the first argument does not suffice to determine the value of the expression: when the first argument of the AND function evaluates to false, the overall value must be false; and when the first argument of the OR function evaluates to true, the overall value must be true. In some programming languages (Lisp), the usual Boolean operators are short-circuit. In others (Java, Ada), both short-circuit and standard Boolean operators are available. For some Boolean operations, like XOR, it is not possible to short-circuit, because both operands are always required to determine the result.

Published By:
Blue Eyes Intelligence Engineering and Sciences Publication (BEIESP)

The short-circuit expression *x* Sand *y* (using Sand to denote the short-circuit variety) is equivalent to the conditional expression if *x* then *y* else false; the expression *x* Sor *y* is equivalent to if *x* then true else *y*.

Short-circuit operators are, in effect, control structures rather than simple arithmetic operators, as they are not strict. ALGOL 68 used "proceduring" to achieve *user defined* short-circuit operators & procedures.

In loosely typed languages which have more than the two truth-values True and False, short-circuit operators may return the last evaluated subexpression, so that *x* Sor *y* and *x* Sand *y* are actually equivalent to if *x* then *x* else *y* and if *x* then *y* else *x* respectively (without actually evaluating *x* twice). This is called "Last value" in the table below.

In languages that use lazy evaluation by default (like Haskell), all functions are effectively "short-circuit", and special short-circuit operators are unnecessary

4.1. Avoiding the execution of second expression's side effects

Usual example.

```
int denom = 0;
if (denom && num/denom) {
 ... // ensures that calculating num/denom never results in divide-by-zero error
}
```

Consider the following example using C language:

```
int a = 0;
if (a && myfunc(b)) {
   do_something();
}
```

In this example, short-circuit evaluation guarantees that myfunc(b) is never called. This is because a evaluates to *false*. This feature permits two useful programming constructs. Firstly, if the first sub-expression checks whether an expensive computation is needed and the check evaluates to false, one can eliminate expensive computation in the second argument. Secondly, it permits a construct where the first expression guarantees a condition without which the second expression may cause a run-time error. Both are illustrated in the following C snippet where minimal evaluation prevents both null pointer dereference and excess memory fetches:

```
   bool is_first_char_valid_alpha_unsafe(const char *p)
{
 return isalpha(p[0]); // SEGFAULT highly possible with p == NULL
}
 bool is_first_char_valid_alpha(const char *p)
{
 return p != NULL && isalpha(p[0]); // a) no unneeded isalpha() execution with p == NULL, b) no SEGFAULT risk
}
```

4.2. Code efficiency

If both expressions used as conditions are simple boolean variables, it can be actually faster to evaluate both conditions used in boolean operation at once, as it always requires a single calculation cycle, as opposed to one or two cycles used in short-circuit evaluation (depending on the value of the first). The difference in terms of computing efficiency between these two cases depends heavily on compiler and optimization scheme used; with proper optimization they will execute at the same speed, as they will get compiled to identical machine code.

Short-circuiting can lead to errors in branch prediction on modern processors, and dramatically reduce performance (a notable example is highly optimized ray with axis aligned box intersection code in ray tracing). Some compilers can detect such cases and emit faster code, but it is not always possible due to possible violations of the C standard. Highly optimized code should use other ways for doing this (like manual usage of assembly code).

4.3. Controlling eagerness in lazy languages

In lazy programming languages such as Haskell, although the default is to evaluate expressions only when they are demanded, it is possible in some cases to make code more eager—or conversely, to make it more lazy again after it has been made more eager. This can be done by explicitly coding something which forces evaluation (which may make the code more eager) or avoiding such code (which may make the code more lazy). *Strict* evaluation usually implies eagerness, but they are technically different concepts.

However, there is an optimisation implemented in some compilers called strictness analysis, which, in some cases, allows the compiler to infer that a value will always be used. In such cases, this may render the programmer's choice of whether to force that particular value or not, irrelevant, because strictness analysis will force strict evaluation.

In Haskell, marking constructor fields strict means that their values will always be demanded immediately. The seq function can also be used to demand a value immediately and then pass it on, which is useful if a constructor field should generally be lazy. However, neither of these techniques implements *recursive* strictness—for that, a function called deepSeq was invented.

Also, pattern matching in Haskell 98 is strict by default, so the ~ qualifier has to be used to make it lazy.

V. CONCLUSION

Demand paging follows that pages should only be brought into memory if the executing process demands them at the time we are increasing the performance we are preferreed to taht lazy evaluation expressions,these evalavation expressions to be considered in various programming languanges.These lazy loading technique also evalate the sum of various exp[ressions.it is easy approach to find out the faults.so mainly this paper focus on the evaluation of expressions with demand paging in virtual memort management.

REFERENCES

[1] Hudak, Paul (September 1989). "Conception, Evolution, and Application of Functional Programming Languages". *ACM Computing Surveys* **21** (3): 383–385. http://portal.acm.org/citation.cfm?id=72554.

[2] Reynolds, John C. (1998). *Theories of programming languages*. Cambridge University Press. ISBN [[Special: BookSources/978052159414|978052159414]].

Published By:
Blue Eyes Intelligence Engineering and Sciences Publication (BEIESP)

http://books.google.com/books?id=HkI01IHJMcQC&pg=PA307

[3] E. P. Markatos and C. E. Chronaki , "A Top-10 approach to prefetching ", Proceedings of INET'98 Geneva, Switzerland, (1998), pp. 276-290.

[4] Y. Jiang, M.Y Wu, and W. Shu, "prefetching : Costs , benefits and performance", Proceedings of the 11th International World Wide Web Conference, New York, ACM, (2002).

[5] A. Venkataramani, P. Yalagandula, and R. Kokku , "The potential costs and benefits of long-term prefetching for content distribution", Computer Communications, 25(4) ,(2002). pp. 367-375.