Polling Efficiently on Stock Hardware

Marc Feeley
Département d'Informatique et Recherche Opérationnelle
Université de Montréal
C.P. 6128, succ. "A", Montréal, Canada H3C 3J7
feeley@iro.umontreal.ca

Abstract

Two strategies for supporting asynchronous interrupts are: the use of the processor's hardware interrupt system and the use of polling. The advantages of polling include: portability, simplicity, and low cost for handling interrupts. Unfortunately, polling has an overhead for the explicit interrupt checks inserted in the code. This paper describes *balanced polling*, a method for placing the interrupt checks which has a low overhead and also guarantees an upper bound on interrupt latency. This method has been used by Gambit (an optimizing native code compiler for Scheme) to support a number of features including multiprocessing and stack overflow detection. The overhead of balanced polling is less than for *call-return* polling which places interrupt checks at every procedure entry and exit. The overhead of call-return polling is typically 70% larger (but sometimes over 400% larger) than the overhead of balanced polling.

1 Introduction

In this paper, the term *interrupt* is defined as an exceptional event for which some special processing is needed (e.g. a heap overflow). The handling of an interrupt is done in three phases. The interrupt is *raised* when the event occurs. At some point after this the processor *detects* the interrupt and then *handles* it by invoking the appropriate interrupt handler. There are two types of interrupts. *Synchronous* interrupts can only be raised at well defined locations in the code (e.g. an arithmetic overflow or invalid pointer dereference). *Asynchronous* interrupts can be raised at any location (e.g. a timer or user interrupt).

1.1 Detecting Interrupts

There are essentially two ways in which interrupts can be detected. They can be detected automatically by the processor's hardware or by explicit checks inserted in the code. These will be called *implicit* and *explicit* detection respectively. On stock hardware, the following interrupts are often detected implicitly: arithmetic overflows, address alignment errors, address translation errors, and user and timer interrupts. The following interrupts are usually detected explicitly: stack and heap overflows, read and write barriers (for incremental [3] and generational [10] garbage collection), and type errors and range errors. This classification depends on the capabilities of the hardware and other implementation constraints. In principle all interrupts could be detected explicitly and it is mainly for performance reasons (but sometimes for simplicity) that they are detected implicitly. Stack overflows can for example be detected implicitly by placing an invalid guard page at the stack's limit. This however may be expensive or overly burdensome to do for a multitasking system which must maintain several small stacks.

There are other reasons why explicit detection may be preferable to implicit detection. Implicit detection is less portable because the trapping mechanism varies from processor to processor. Portability can be increased somewhat by relying on standard libraries such as the UNIX `signal.h` routines but this increases the cost of each interrupt and it only gives access to the most common types of interrupts.

Another problem with implicit detection on stock hardware is the high cost of handling an interrupt. Current trap architectures enter kernel mode to process the interrupt. This requires that the processor's state (all of the registers or some significant subset) be saved and later restored. If the trap handler is in user space, two additional transitions between user and kernel space are needed. Explicit detection has a much lower cost. The main reasons are that control remains in user space, the call to the interrupt handler can be open coded and/or specialized, and a minimum number of registers have to be saved (since the compiler knows which registers are live and possibly which are clobbered by the handler). Unfortunately, the cost of detecting the interrupt (T_{detect}) applies to all potential interrupt points. Explicit detection will only be more efficient than implicit detection if interrupts are sufficiently frequent to make the following inequality true

$$T_{detect} < p(T_{implicit} - T_{explicit})$$

where p is the probability that an interrupt is detected, and $T_{implicit}$ and $T_{explicit}$ are respectively the times to handle the interrupt in each approach. As a concrete example for synchronous interrupts, assume that an arithmetic operation has to be checked for overflow and that a single "branch on overflow instruction" can detect this (i.e. $T_{detect} = 1$). Assuming $T_{implicit} - T_{explicit}$ is 100 instructions[1], explicit

[1] Johnson [8] reports a cost of 106 instructions on the SPARC to take an overflow trap on the TADDccTV instruction. This cost

detection will be preferable if $p > .01$, that is if there is an overflow at least once out of every 100 times the arithmetic operation is executed.

1.2 Asynchronous Interrupts

The handling of asynchronous interrupts is an issue in many languages and systems but this paper looks at the issues in the context of a Lisp system because Lisp stresses many problems (e.g. garbage collection and small functions). The methods described here are however applicable to other languages.

Because they can be raised at any point in the program, asynchronous interrupts pose special problems.

1.2.1 Critical Sections

To support garbage collection, Lisp systems must maintain a memory state that can be parsed by the garbage collector. However, the system's state can temporarily become inconsistent in the middle of some code sequences. Storing a 64 bit pointer into x might for example translate into a sequence of two 32 bit store instructions. Since immediately after the first store x does not contain a valid pointer, the system must insure that x is not accessed, either explicitly or by the garbage collector, before the second store is performed. Other cases where the system potentially enters an inconsistent state include: updating system structures, saving Lisp objects in locations not scanned by the garbage collector, and saving non Lisp objects in locations scanned by the garbage collector (e.g. when clearing or extracting a pointer's type tag, and "unboxing" numbers). Interrupts are a problem because they can not be processed in an inconsistent state if the interrupt handler might call the garbage collector or otherwise access the part of the state which is inconsistent. This is a definite possibility if the interrupt handlers are written in unrestricted Lisp or if the system supports user interrupts (for entering a break loop) or preemption interrupts (for multitasking). Either the system is carefully designed to never enter inconsistent states (which precludes a number of compiler optimizations), or the code sequences are protected within uninterruptible critical sections[2].

When interrupts are detected implicitly, a simple implemention of critical sections is to inhibit interrupts for the duration of the critical section. This can be done by surrounding it with a pair of instructions to disable and enable the interrupts. Interrupts raised during a critical section will get handled when the interrupts are reenabled at the end of the critical section. This approach suffers from a high overhead for the added disable/enable pairs but the overhead can be reduced somewhat by using a single pair around a grouping of critical sections.

Techniques that completely avoid the overhead of disable/enable pairs do exist. Maclisp and Lucid Common Lisp keep the location of all critical sections in a table. When an interrupt is raised, the address of the interrupted instruction is looked up in the table. If the interrupt was raised inside a critical section, the rest of the critical section is first executed and then the interrupt handler is called. Maclisp achieves this by single stepping the code to the end of the critical section (by the use of the PDP-10's XCT instruction)[3]. Lucid Common Lisp uses this strategy for the internal subroutines called by compiled code [14]. T [9] has a slightly different zero cost strategy that does not use tables. When an interrupt is raised, the continuation is modified so that soon after the critical section is finished the interrupt handler is called. Another strategy [1], designed specifically for the problem of allocation in a multitasking system, also single steps the interrupted code but not on every interrupt. Instead, it is the garbage collector which single steps each task if its program counter indicates that it is in the middle of an allocation.

When interrupts are detected explicitly, critical sections are easy to implement. The compiler simply has to insure that interrupt checks are only generated in "safe" places (i.e. outside critical sections).

2 Polling

The biggest issue with explicit detection of asynchronous interrupts is the placement of the interrupt checks. Placing them after every instruction (outside critical sections) would clearly be too expensive. A more efficient approach consists of putting interrupt checks at specific places in the code so that interrupts are checked, or *polled*, periodically.

A few systems use polling for asynchronous interrupts (e.g. Gambit [5], MIT-Scheme [12], MultiScheme [11], Allegro Common Lisp, and SML/NJ [2]). Gambit is a Scheme system with multiprocessing extensions taken from Multilisp [7]. Asynchronous interrupts are used to distribute work among the processors by dynamic partitioning. Idle processors send "work request" interrupts to other processors which must respond with a task to run or no task. Polling was chosen because of the high frequency of work request interrupts when programs use fine grain parallelism and because it is important to answer work requests quickly to minimize the idleness of the processors.

In theory, the compiler could arbitrarily reduce the polling overhead (O_{poll}) by decreasing the proportion of interrupt checks executed with respect to the normal instructions executed by the program. If all instructions take unit time then $O_{poll} = N_{poll}/N_{instr}$, where N_{poll} is the number of interrupt checks executed and N_{instr} is the number of normal instructions executed. This strategy lowers the frequency of interrupt checking and consequently increases the average latency of interrupts (i.e. the time between the raising of the interrupt and the handling of the interrupt). Average latency (L) and overhead are inversely related by

$$L = \frac{N_{poll} + N_{instr}}{N_{poll}} = 1 + \frac{1}{O_{poll}}$$

Here latency is expressed in number of instructions. To account for non-unit time instructions, latency can be expressed in units of time (or number of machine cycles). This leads to the definitions

$$O_{poll} = \frac{T_{poll}}{T_{instr}}$$

does not account for the processing of the overflow, so $T_{implicit} - T_{explicit}$ will be roughly 100 instructions.

[2] Some systems support user specified critical sections. However, only critical sections that span a few instructions are considered here since this functionality is sufficient to support user specified critical sections.

[3] The actual mechanism used in MacLisp is even more general [13]. It allows annotations on the critical sections that specify whether interrupts should: be ignored, be deferred to the end of the critical section, restart the critical section, or invoke a special handler.

$$L = \frac{T_{poll} + T_{instr}}{N_{poll}}$$

where T_{poll} is the total time spent on interrupt checks and T_{instr} the time spent on normal instructions. If an interrupt check takes k time units on average then $L = k(1 + \frac{1}{O_{poll}})$. To simplify the discussion, all instructions will be assumed to take unit time.

3 The Problem of Procedure Calls

Although polling seems simple enough to implement, there is a complication. Normally, programs are not composed of a single stream of instructions. If this were the case the compiler could simply count the instructions it emits and insert an interrupt check after every so many instructions. Branches and procedure calls can alter the flow of control in unpredictable ways and so, it isn't clear how the compiler can achieve a constant number of instructions between interrupt checks. A reasonable compromise is to ask of the compiler to emit interrupt checks such that a given latency (L_{max}) is never exceeded.

3.1 Code Structure

Before exploring the problem further, it is convenient to introduce a formalism to describe the structure of a procedure's code. The code of a procedure will be represented by a graph of basic blocks of instructions. There are two special types of basic blocks: *entry points* and *return points*. A procedure has a single entry point and one return point for each procedure call in subproblem position.

Branches are only allowed as the last instruction of a basic block. Four types of branches exist: local branches (possibly conditional) to other basic blocks of the same procedure, tail calls to procedures (i.e. reductions), non-tail calls to procedures (i.e. subproblems), and returns from procedures. Local branches and non-tail calls are not allowed to form cycles and thus they impose a DAG structure to the code. Following Scheme's convention, it is assumed that loops are always expressed with tail calls.

Note that subproblem and reduction calls always jump to entry points and that procedure returns always jump to return points. These restrictions are important to remember because they simplify the analysis of the control flow.

Figure 1 gives the graph for the procedure `for-each` which contains all four types of branches. Returns and tail calls have been represented with dotted lines because they do not correspond to DAG edges. Solid lines are used for subproblem calls to highlight the fact that, just like local branches, it is known where control continues after the procedure returns (if it returns at all). The generality of the DAG is only needed to express the sharing of code. For the moment, it is sufficient to make the simplifying assumption that the DAG has been converted into a tree by duplicating each shared branch. The handling of shared code is described in Section 6.

A necessary condition for any polling strategy is that an inline sequence of more than L_{max} instructions is never generated without an intervening interrupt check. The compiler can exploit the code structure for this purpose. A *locally connected section* is any subset of the basic blocks that is connected by local branches only (for example, the three basic blocks at the top of Figure 1 or the bottom one). For any instruction I in a locally connected section, it is easy

```
(define (for-each f l)
  (if (null? l)
    #f
    (begin
      (f (car l))
      (for-each f (cdr l)))))
```

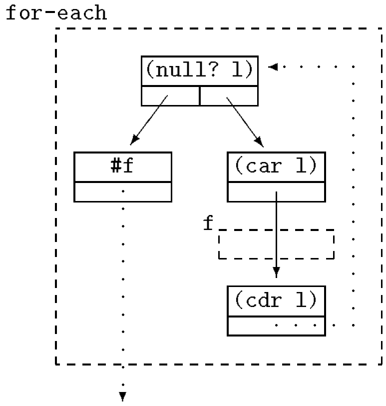


Figure 1: The `for-each` procedure and its corresponding code graph.

to determine what instructions are on the path to I from the section's root. These instructions are exactly those that are executed at runtime before I. Thus, for any instruction in a locally connected section, the compiler can tell how far back the last interrupt check occured (assuming there is one on the same path from that section's root). The number of instructions that seperate an instruction from the previous interrupt check is called the instruction's *delta*. For instructions that are not preceded by an interrupt check in the same section, the definition of delta will vary according to the polling strategy. When the delta reaches L_{max}, an interrupt check is inserted by the compiler before the instruction. If this is in the middle of a critical section, the compiler must move the interrupt check to the end (or beginning) of the critical section.

3.2 Call-Return Polling

Polling strategies differ in how the transition between locally connected sections is handled. *Call-return* polling is a simple strategy that consists of putting an interrupt check as the very first instruction of each section's root. Since the root of a section is either the entry point of the procedure or the return point of a subproblem call, this corresponds to polling on procedure call and return.

At first glance, polling on return does not seem necessary since a procedure is bound to get called at some point. However, this would mean that there is no upper bound on the interrupt latency because it is possible to build a continuation that does not call any procedure for an arbitrarily long time. For example, the following procedure does not call any procedure during the unwinding of the recursion.

```
(define (length-even? lst)
  (if (and (pair? lst) (length-even? (cdr lst)))
    #f
    #t))
```

```
(define (make-person name age gender)
  (vector name age gender))

(define (person-name   x) (vector-ref x 0))
(define (person-age    x) (vector-ref x 1))
(define (person-gender x) (vector-ref x 2))

(define (sum vect l h) ; sum vector from l to h
  (if (= l h)
    (vector-ref vect l)
    (let* ((mid (quotient (+ l h) 2))
           (lo  (sum vect l mid))
           (hi  (sum vect (+ mid 1) h)))
      (+ lo hi))))
```

Figure 2: Two instances of short lived procedures.

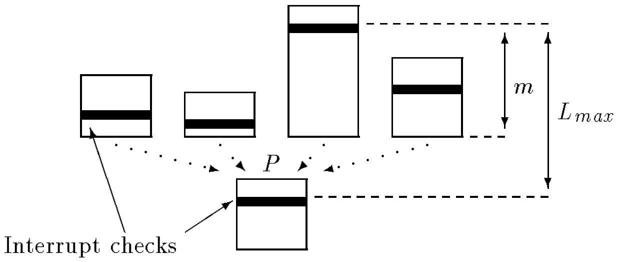


Figure 3: The maximal delta method.

There are a few variations of call-return polling. The interrupt check at the return point can be removed if checks are put on all return branches. Similarly, the interrupt check at the entry point can be replaced by checks on branches to procedures (both tail calls and non-tail calls). The four possible variations give equivalent dynamic behavior (i.e. same number of interrupt checks executed) but one may be preferable to the others if it yields more compact code. This depends on the particular code generation techniques used by the compiler and the programs being compiled. Compactness of code is not a big issue here so it won't be considered further.

4 Short Lived Procedures

Unfortunately, call-return polling has poor performance in certain circumstances. The worst case occurs when procedures are short lived, that is they return shortly after being called. At least two interrupt checks are performed per procedure call in subproblem position (once on entry and once on exit) and one if it is a reduction. This is a significant overhead if the procedure contains few instructions. In languages that promote the use of large procedures this would not be a serious problem, but in Lisp it is common to structure programs into several short procedures.

Two instances of this style, typified by the procedures in Figure 2, are the implementation of data abstractions and divide and conquer algorithms. In binary divide and conquer algorithms, at least half of the recursive calls correspond to the base case. If the algorithm is fine grained, such as the procedure `sum`, the overhead of polling will be noticeable because all the leaf calls are short lived.

The problem with call-return polling is that it doesn't take the structure of the program into account. If it is known that a procedure P is always called when delta is equal to n, then the compiler could infer that upon entry to P, delta is n. This would introduce a "grace period" of $L_{max} - n$ instructions at P's entry point during which interrupt checks are not needed. A similar statement holds for return points. Note that this yields a perfect placement of interrupt checks if it is carried out at all procedure entry and return points. Interrupt checks would occur exactly every L_{max} instructions.

A more realistic solution is needed to handle the case where procedures and return points are called in different contexts. A simple extension to the previous method is to use m instead of n, where m is the maximum delta of all call sites to P (and similarly for return points). This "maximal delta" method is illustrated in Figure 3 where dark rectangles are used to represent interrupt check instructions. This is not an ideal solution for two reasons. First, it forces all control paths through P to have an early interrupt check (in P) if just one call site to P has a high delta. It would be much better if each procedure call "paid it's own way", meaning that interrupt checks should be put on the call sites with high deltas. Not only would this improve P's grace period, it would put the interrupt check where it causes the least overhead (because a high delta is a sign of a high number of normal instructions)[4].

A second shortcoming of this solution is that the source and destination of procedure calls has to be known at compile time. This information is not generally available. One could reasonably argue that with the use of programmer annotations and/or control flow analysis the destination of most procedure calls could be inferred by the compiler for typical programs. However, the destination of returns is harder to determine because it would require a full dataflow analysis of the program and in general a procedure has a number of possible points to return to. The existence of higher order functions is another source of difficulty.

5 Balanced Polling

This section presents a general solution that does not rely on any knowledge of the control flow of the program. The method could be extended with appropriate rules, such as maximal delta, to better handle the cases where control flow information is available, but this is not considered here.

The idea is to define polling state invariants for procedure entry and exit. The polling strategy expects these invariants to be true at the entry and return points of all procedures and consequently must arrange for them to be true at procedure calls and returns.

Specifically, the invariant at procedure entry is that interrupts have been checked at most $L_{max} - E$ instructions ago. Here E is the grace period at entry points and is constant for all procedures. In other words, delta is defined to be $L_{max} - E$ at entry points. Delta now represents an upper bound to the distance from the last interrupt check. The invariant at procedure return is more complex. Either delta is less than E or, the path from the entry point to the return instruction is at most E instructions. These invariants are shown in Figure 4. Procedure P has two branches that illustrate the two cases for procedure return. Note that a procedure can be exited by a procedure return as well as

[4] This assumes that all paths to P are equiprobable.

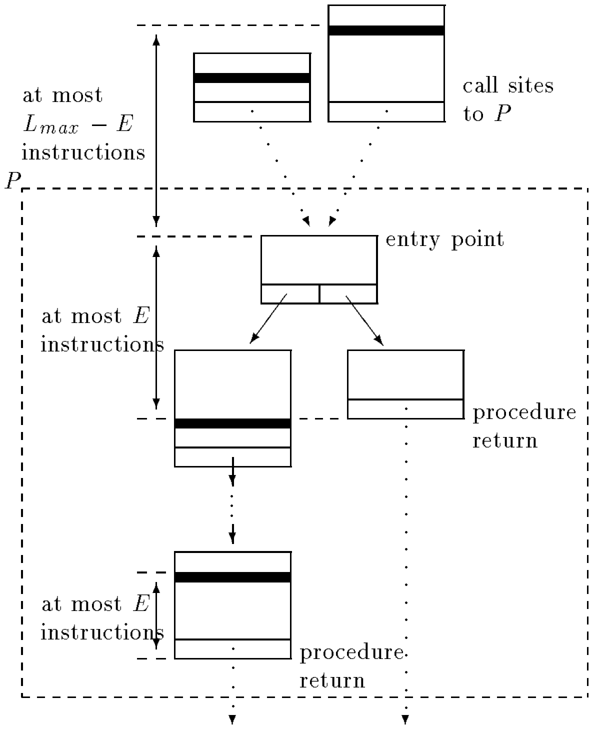


Figure 4: Procedure return invariants in balanced polling.

a reduction call. For now, reduction calls will be ignored to simplify the discussion.

5.1 Subproblem Calls

These invariants have important implications. To begin with, short lived procedures are handled well because there is no need to check interrupts on any path that returns shortly without a call to another procedure (i.e. with less than E non-call instructions). This corresponds to the right-most path in Figure 4.

Moreover it allows the delta at return points to be defined as E plus the delta for the corresponding call point. This can be confirmed by considering the two possible cases. Assume procedure P_1 does a subproblem call to procedure P_2 which eventually returns back to P_1 via a procedure return in P_2:

$$P_1 \xrightarrow[\text{call}]{\text{subproblem}} P_2 \xrightarrow[\text{return}]{\text{procedure}} P_1$$

Either the last interrupt check was in P_2, so by definition delta at the return point (in P_1) is less than E. Alternatively, P_2 was short lived and didn't check interrupts, so there are at most E instructions that seperate the call site (in P_1) from the return point (in P_1). As far as polling is concerned, a procedure called in subproblem position can be viewed as an interrupt check free sequence of E instructions. The compilation rule here is that if delta at a call point exceeds $L_{max} - E$ then an interrupt check is inserted at the call.

This rule means that up to $\lfloor L_{max}/E \rfloor$ subproblem procedure calls can be done in sequence without any interrupt checking. To see why, consider the scenario where the first call is immediately preceded by an interrupt check. At the return point, delta is equal to E. If the instructions for argument setup and branch are ignored, delta at the n^{th} return point is $n \times E$. Only when this reaches L_{max} is an interrupt check needed.

5.2 Reduction Calls

As described, the polling strategy does not handle reduction procedure calls (tail calls) very gracefully. The case to consider here is when a subproblem call is to a procedure which exits via a series of tail calls, finally ending in a procedure return:

$$P_1 \xrightarrow[\text{call}]{\text{subproblem}} P_2 \xrightarrow[\text{call}]{\text{reduction}} P_3 \cdots P_{n-1} \xrightarrow[\text{call}]{\text{reduction}} P_n \xrightarrow[\text{return}]{\text{procedure}} P_1$$

An interrupt check must always be put at a reduction call point to guard against the case where the called procedure returns shortly without checking interrupts (as in P_{n-1} calling P_n). Note that the return point in P_1 can have a delta as low as E. Note also that P_n can execute as many as E interrupt check free instructions before returning to the return point in P_1. Thus, it is not valid for P_{n-1} to jump to P_n with a delta greater than 0 because this might violate the polling invariant at the return point in P_1.

The treatment of reductions can be improved by introducing a new parameter (R) and consequently adjusting the polling invariants to support it. R is defined as the largest admissible delta at a reduction call. Thus, an interrupt check is put on any reduction call whose delta would otherwise be greater than R. The same polling behavior as before is obtained by setting R to 0. The polling constraints for reduction calls can be relaxed by increasing the value of R. R is at most $L_{max} - E$ because a reduction call might be to a procedure that doesn't check interrupts for as many as E instructions.

A new invariant for return points has to be formulated to accomodate R. The delta at return points must now be at least $E + R$ to account for the case explained previously. That is, on return to P_1 there could be up to E instructions in P_n plus as much as R instructions at the tail of P_{n-1} since the last interrupt check. When the compiler encounters a subproblem procedure call it sets the delta at the return points to E plus the largest value between R and the delta for the corresponding call point. Of course, if this value is greater than L_{max} an interrupt check is first put at the call site and the delta at the return point is set to $E + R$. The introduction of R also makes it possible to relax the invariant for procedure returns. Since the delta for return points is at least $E + R$, a procedure return's delta as high as $E + R$ can be tolerated without requiring an interrupt check. With these new invariants, there can be up to $\lfloor (L_{max} - R)/E \rfloor$ subproblem procedure calls in sequence without interrupt checks. This polling strategy will be called *balanced polling*. A summary of the compilation rules for balanced polling is given in Figure 5.

The two constants E and R must be chosen carefully to achieve good performance. Small values for E and R increase the number of interrupt checks for short lived procedures and tail recursive procedures respectively. On the other hand, high values increase the number of interrupt checks in code with many subproblem procedure calls (e.g. recursive procedures). Choosing $E = R = \lfloor L_{max}/k \rfloor$ is a

Location	Action
Entry point	$\Delta \leftarrow L_{max} - E$
Non-branch instruction	**if** ($\Delta \geq L_{max} - 1$) **then** add interrupt check $\Delta \leftarrow 0$ $\Delta \leftarrow \Delta + 1$ (for next instruction)
Subproblem call	**if** ($\Delta \geq L_{max} - E$) **then** add interrupt check $\Delta \leftarrow 0$ $\Delta \leftarrow E + \max(R, \Delta)$ (for return point)
Reduction call	**if** ($\Delta \geq R$) **then** add interrupt check
Procedure return	**if** ($\Delta \geq E + R$) **and** there are interrupts on path from entry point **then** add interrupt check

Figure 5: Compilation rules for balanced polling.

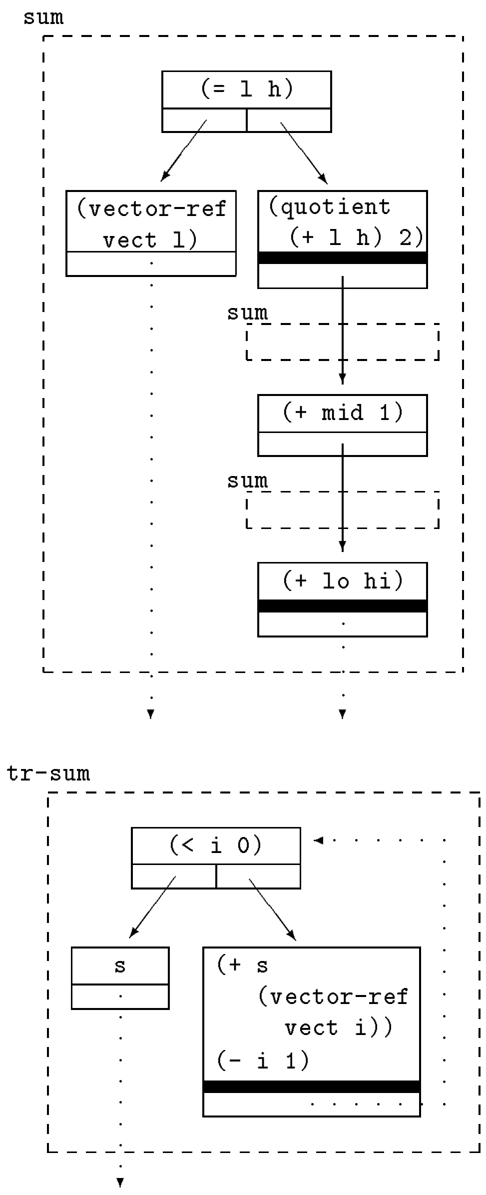
reasonable compromise and a value of $k = 6$ gives good performance in practice. This suggests that there are typically less than 6 subproblem procedure calls per procedure in the benchmark programs (see Section 8).

5.3 Minimal Polling

The choice of L_{max} is also an issue. A high L_{max} will give a low polling overhead. However, it is important to realize that there is a limit to how low the polling overhead can be made by increasing the value of L_{max}. This is due to the conservative nature of the strategy. Whatever the values of L_{max}, E and R are, at least one interrupt check is generated between the entry point and the first procedure call. Delta is $L_{max} - E$ on entry to a procedure, so clearly the first call (reduction or subproblem) must be preceded by an interrupt check. Similarly, there is at least one interrupt check between any return point and the exit of the procedure (return or reduction call) because delta at any return point is at least $E + R$. These two types of paths are the only ones that are necessarily part of any unbounded length path. Thus, it is sufficient to have one interrupt check on each of these paths to guarantee that all possible control paths have a bounded number of instructions between interrupt checks. This *minimal polling* strategy is useful because its overhead is a lower bound that can be used to evaluate other techniques.

An example of minimal polling for the procedure `sum` and the tail recursive variant `tr-sum` is presented in Figure 6. For the call (`sum v` l h) there are exactly $2 \times (h - l)$ interrupt checks executed or nearly one interrupt check per procedure call (assuming $h - l + 1$ is a power of two). By comparison, checking interrupts at procedure entry and exit would require two interrupt checks per procedure call. However, for the tail recursive procedure `tr-sum` both methods are essentially equivalent with one interrupt check per iteration.

It is interesting to note that balanced polling is more




```
(define (tr-sum vect s i)
  (if (< i 0)
    s
    (tr-sum
      vect
      (+ s (vector-ref vect i))
      (- i 1))))
```

Figure 6: Minimal polling for the recursive procedure `sum` and a tail recursive variant.

general than minimal polling and call-return polling. These can be emulated by judiciously choosing E, R and L_{max}. Minimal polling is obtained when $0 \ll E = R \ll L_{max}$ (i.e. E and R are arbitrarily large and L_{max} is arbitrarily larger). An interrupt check is put at the first call and another one is put at the return or reduction call that follows the last return point. Call-return polling occurs when $0 = E \ll R = L_{max}$. This places interrupt checks at all entry points and return points.

6 Handling Join Points

It has been assumed that the code of procedures is in the form of a tree. However, the compilation of conditionals (i.e. `and`, `or`, `if` and `cond`) in subproblem position introduces join points that give a DAG structure to the code. Certain optimization techniques, such as common code elimination, can also produce join points to express the sharing of identical code branches. Join points can be handled with the maximal delta method. That is, the delta at the join point is the maximum delta of all branches to the join point.

7 Polling in Gambit

Polling is a general mechanism that can serve many purposes. Gambit is a Lisp system for shared memory multiprocessors. Gambit uses polling for

- Stack overflow detection
- Preemption interruption (for multitasking)
- Inter-processor communication (for work distribution)
- Inter-task communication (for killing tasks)
- Barrier synchronization (e.g. for synchronizing all processors for a garbage collection and to copy objects and load code to the private memory of every processor)

A special technique is used to check all these cases with a single test. The interrupt flag is really a pointer that is normally set to point to the end of the area available for the stack. Note that because stack overflows are detected asynchronously, the stack extends a little further than this limit. An interrupt check consists of comparing the flag to the current stack pointer, and to jumping to an out of line handler when the stack pointer exceeds the limit. A processor can be interrupted by setting the flag to a value that forces this situation (e.g. 0). The interrupt handler then uses some other flags to discriminate between the possible sources of interrupt. SML/NJ [2] uses a similar approach based on the heap allocation pointer.

Although it can be done with a single test, the interrupt check may still be relatively expensive. Because all processors must have access to a processor's interrupt flag, it is located in shared memory (which can't be cached easily). Increasing L_{max} is not a viable solution because the polling frequency can't be lowered beyond a certain point. To provide a finer level of control, interrupts can be checked intermittently. Polling instructions generated by the compiler represent "virtual" interrupt check points and an actual check of the interrupt flag occurs once every n virtual checks. This is easily implemented by a private counter that is decremented at every virtual check. When it reaches zero it is reset to n and the interrupt check is performed. The average cost of an interrupt check will thus be the cost of updating and checking the private counter plus $1/n^{th}$ the cost of checking the interrupt flag.

An interesting optimization occurs here. Balanced polling has a tendency to put the interrupt checks at branch points. An interrupt check itself involves a branch instruction so in many cases it is possible to combine the two branches into a single one. Moreover, several machines have a combined "decrement and branch" instruction that helps reduce the cost even further. All these ideas are implemented in Gambit. Below is an example showing the M68020 assembly code generated by Gambit for the tail-recursive procedure `last`. The boxed part is the interrupt check sequence (note that $n = 10$ and that register `a5` always contains a pointer to the stack limit pointer).

```
(define (last lst)
  (let ((rest (cdr lst)))
    (if (pair? rest)
      (last rest)
      (car lst))))
```

```
L1:               ; entry: d1=lst & a0=ret adr

  movl  d1,a1     ; rest <- (cdr lst)
  movl  a1@-,d2

  btst  d2,d7     ; (pair? rest)
  bne   L2

  movl  d2,d1     ; lst <- rest

  dbra  d5,L1     ; decrement and test counter
  moveq #9,d5     ; reset counter
  cmpl  a5@,sp    ; is sp beyond the limit?
  bcc   L1        ; no interrupt if sp>=limit
  jsr   handler   ; call interrupt handler

  bra   L1        ; reduction call to last

L2:
  movl  d1,a1     ; result <- (car lst)
  movl  a1@,d1

  jmp   a0@       ; return
```

8 Results

To have a better idea of the polling overhead that can be expected from these polling methods, it is important to measure the overhead on actual programs. Two situations are especially interesting to evaluate: the overhead on typical programs and on pathological programs that are meant to exhibit the best and worst performance.

Several programs and polling methods were tested. The programs were compiled by Gambit in four different ways: with no interrupt checks, with minimal polling, with call-return polling and balanced polling. For balanced polling, L_{max} was set to values from 10 to 90 and E and R were set at $\lfloor L_{max}/6 \rfloor$. A value of $n = 10$ was used for polling intermittently. The average runtime on ten runs was taken for each situation. The overhead of minimal polling over the program compiled with no interrupt checks is reported in the first column of Table 1. The overhead for the other polling methods is expressed relative to the overhead of minimal polling. Thus a relative overhead of 2 means that the overhead is twice that of minimal polling.

The program `tight`, shown below, was designed to exhibit worst-case behavior.

```
(define (tight n)
  (if (> n 0)
    (tight (- n 1))))
```

Program	Minimal polling O_{poll} (%)	Call-return polling Rel. ov.	Balanced polling: Rel. ov. when $E = R = \lfloor L_{max}/6 \rfloor$ and L_{max} is 10	20	30	40	50	60	70	80	90
`tight`	83.9	1.0	2.5	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
`unfolded`	6.1	0.9	10.8	6.5	4.2	3.5	3.7	3.9	2.3	2.3	2.3
`boyer`	21.5	1.4	1.7	1.1	1.0	1.0	1.1	1.0	0.9	0.9	0.9
`browse`	14.7	1.1	1.6	1.1	0.8	1.0	1.7	1.2	1.0	1.0	0.9
`cpstak`	10.9	1.2	1.9	1.5	1.2	1.0	1.1	1.0	1.0	1.0	1.1
`dderiv`	9.0	1.6	2.1	1.4	1.6	1.2	1.0	1.3	1.3	1.2	1.3
`deriv`	8.1	1.4	1.8	1.4	1.8	1.1	0.9	1.0	1.0	1.1	1.2
`destruct`	21.3	1.1	2.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
`div`	14.1	1.0	1.3	0.9	1.0	1.0	1.0	1.0	1.0	1.0	1.0
`puzzle`	14.5	0.9	2.1	1.7	1.2	1.0	1.0	1.0	0.9	0.9	0.9
`tak`	8.7	4.6	3.9	1.4	1.8	1.2	1.0	1.0	1.0	1.0	1.0
`takl`	29.3	0.9	1.5	1.0	1.1	1.0	0.9	0.9	0.9	0.9	0.9
`traverse`	16.9	1.5	2.5	1.3	0.9	0.9	0.9	0.9	0.9	0.9	0.9
`triangle`	3.9	3.7	6.0	6.0	3.2	3.8	2.4	2.1	2.3	1.0	2.0
`compiler`	14.4	1.8	2.3	1.3	1.1	1.0	1.0	1.0	1.1	1.0	1.0
`conform`	10.5	2.5	2.8	1.7	1.3	1.1	1.2	1.4	1.3	1.4	1.2
`earley`	6.4	1.5	2.3	1.6	1.5	1.0	1.1	2.1	0.8	1.1	1.2
`peval`	9.7	1.7	2.2	1.5	1.0	1.1	1.1	1.3	1.0	1.0	1.1
`abisort`	11.4	1.3	2.5	1.7	1.4	1.4	1.0	1.1	1.1	1.0	1.0
`allpairs`	4.4	1.0	3.9	2.6	2.0	2.0	2.0	0.5	1.8	1.0	1.0
`fib`	18.7	2.1	2.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
`mm`	4.7	1.1	3.0	2.7	3.0	1.6	2.2	2.2	0.8	0.9	0.9
`mst`	10.5	1.6	2.2	1.5	2.1	1.0	1.2	1.0	0.8	1.1	1.0
`qsort`	12.3	1.3	1.9	1.3	1.0	1.3	1.0	1.0	1.0	1.0	1.0
`queens`	15.2	1.4	3.0	1.5	1.5	1.5	1.4	1.3	1.3	1.2	1.3
`rantree`	11.4	2.5	2.2	1.2	0.9	1.4	1.1	1.3	1.0	1.0	0.9
`scan`	6.6	2.4	3.5	2.0	0.8	0.8	1.2	1.0	1.0	1.0	1.0
`sum`	11.8	1.8	2.5	1.4	0.7	0.5	0.9	1.0	0.8	0.9	0.8
`tridiag`	1.6	2.7	7.9	4.5	4.3	4.2	3.4	3.7	3.9	3.0	3.6

Table 1: Overhead of polling methods on benchmark programs.

It is a tight loop that doesn't do anything except update a loop counter. There are only two instructions executed on every iteration: an increment and a conditional branch. Interrupt checks will clearly add a high overhead to this. For most polling methods the overhead is about 80%. In the case of balanced polling with $L_{max} = 10$ the overhead is roughly twice that because two interrupt checks get added to every loop.

The program `unfolded` is the same loop as `tight` but unfolded 80 times. Thus, it is a long inline sequence of 80 decrements followed by one conditional branch instruction. The polling methods do well on this program (about 6% for minimal and call-return polling) because procedure calls are relatively infrequent and it is easy to handle the inline sequence of instructions. As expected for balanced polling, increasing L_{max} decreased the overhead, down to about 14%. L_{max} would have to be greater than 90 to reduce the overhead to that of minimal polling (at $L_{max} = 90$ there are two interrupt checks per loop).

The other programs are split in three groups. The first group contains programs from the Gabriel benchmark suite [6]. The second group contains more realistic applications that have not been designed with benchmarking in mind. They are fairly sizeable with at least 500 lines of code (15,000 for `compiler`). The last group contains parallel programs taken from [4]. These programs are written in Multilisp but were compiled as sequential programs (i.e. with `FUTURE` and `TOUCH` operations removed) to factor out the overhead of supporting parallelism.

The results show that minimal polling outperforms call-return polling in nearly all cases. Sometimes by as much as a factor of four, but by a factor closer to 1.7 on average. The largest differences occur for fine grain recursive programs (e.g. `tak` and `fib`) and programs with a profusion of data abstraction procedures (e.g. `conform`). The performance of balanced polling is rather poor for small values of L_{max}, two to three times the overhead of minimal polling when $L_{max} = 10$. However, balanced polling gives performance close to minimal polling when L_{max} is high. With $L_{max} = 90$ the overhead ranges from 5% to 25%. The highest overheads are for fine grain recursive programs. Overall, the average overhead for balanced polling is about 15% for values of L_{max} higher than 50.

Acknowledgements

I wish to thank the following people for their help in understanding the interrupt handling mechanism of particular systems: Guy L. Steele Jr. and Jonathan Rees for Maclisp; Jon L. White for Maclisp and Lucid Common Lisp; David Kranz for T; and Guillermo J. Rozas for MIT-Scheme. I also wish to thank the reviewers for their help.

References

[1] A. W. Appel. Allocation without locking. *Software Practice and Experience*, 19(7):703–705, July 1989.

[2] A. W. Appel. *Compiling with continuations.* Cambridge University Press, 1992.

[3] H. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[4] M. Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared Memory Multiprocessors.* PhD thesis, Brandeis University Department of Computer Science, 1993.

[5] M. Feeley and J. S. Miller. A parallel virtual machine for efficient Scheme compilation. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990.

[6] R. P. Gabriel. *Performance and Evaluation of Lisp Systems.* Research Reports and Notes, Computer Systems Series. MIT Press, Cambridge, MA, 1985.

[7] R. Halstead. Multilisp: A language for concurrent symbolic computation. In *ACM Trans. on Prog. Languages and Systems*, pages 501–538, October 1985.

[8] D. Johnson. Trap architectures for Lisp systems. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990.

[9] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. Orbit: An optimizing compiler for Scheme. In *ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233, June 1986.

[10] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[11] J. S. Miller. Implementing a Scheme-based parallel processing system. *International Journal of Parallel Processing*, 17(5), October 1988.

[12] G. J. Rozas. Liar, an Algol-like compiler for Scheme. S. b. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1984.

[13] G. L. Steele Jr. Private Communication, December 1992.

[14] J. L. White. Private Communication, December 1992.