Trap Architectures for Lisp Systems

Douglas Johnson
Texas Instruments MS 238
P.O. Box 655474
Dallas, Texas 75265
johnson@ti.com
(214) 995-0343

November 18, 1988

Abstract

Recent measurements of Lisp systems show a dramatic skewing of operation frequency. For example, small integer (fixnum) arithmetic dominates most programs, but other number types can occur on almost any operation. Likewise, few memory references trigger special handling for garbage collection, but nearly all memory operations could trigger such special handling. Systems like SPARC^{tm} and SPUR have shown that small amounts of special hardware can significantly reduce the need for inline software checks by trapping when an unusual condition is detected.

A system's trapping architecture now becomes key to performance. In most systems, the trap architecture is intended to handle errors (e.g., address faults) or conditions requiring large amounts of processing (e.g., page faults). The requirements for Lisp traps are quite different. In particular, the trap frequency is higher, processing time per trap is shorter, and most need to be handled in the user's address space and context.

This paper looks at these requirements, evaluates current trap architectures, and proposes enhancements for meeting those requirements. These enhancements increase performance for Lisp 9%–32% at cost of about 1.4% more CPU logic.

1 Introduction

Much of the early RISC design work focused on the features needed to support static languages (particularly C) well. In static languages, the data types are known at compile time and any runtime storage management is done explicitly by the programmer. With careful design and cost analysis, RISC architectures are providing dramatic performance improvements over earlier architectures [MIPS 86].

There is increasing interest in dynamic languages such as Smalltalk or Lisp. Dynamic language data types may not be known until runtime and storage management is done implicitly (i.e. garbage collection). These languages are useful in applications ranging from artificial intelligence to user interfaces. The boundary between static and dynamic languages blurs as C++ acquires more dynamic characteristics.

Cursory analysis of dynamic languages shows that RISC architectures are well matched to the majority of dynamic language operations. Steenkiste showed that Lisp programs are dominated by data movement, function calling, and integer arithmetic [Steenkiste 88]. Ungar had similar results for Smalltalk [Ungar 86]. Typical RISC architectures are optimized for those operations. The problem in supporting dynamic languages well is one of possibilities rather than probabilities.

For example, any given arithmetic operation will probably have simple integer operands, but those operands may be any type. In the general case, one cannot know the operands' types ahead of time (e.g. at compile time). Therefore, it is necessary to do extensive checking at runtime, which can be expensive in time and code space.

Data movement presents similar problems. An incremental garbage collection system [Baker78] requires a *read barrier*[1] that must be checked on every fetch from heap storage. Generational garbage collectors require a *write barrier*[2] that must be checked with every store into the heap [Lieberman 83]. So far, the cost of inline code for a read barrier has prevented practical implementation of incremental garbage collection on conventional processors. Inline checks are used for write barriers on conventional processors. They are practical because stores are much less frequent than reads, but the checks

[1]A *read barrier* is used to copy objects as they are referenced. Basically, it requires some special processing whenever a pointer to an uncopied object is fetched.

[2]A *write barrier* is used to remember pointers to recently created objects so those objects may be easily collected. It requires special processing whenever a pointer to a "young" object is stored in an "old" object.

still cause a significant performance degradation.

Efficient implementation of Lisp and other dynamic languages on RISC processors requires dramatically different implementation strategies than were used on Lisp machines [Bosshart 87,Moon 87]. Those machines used extensive special hardware and microcode to do the necessary checking and special case handling. RISC processors do not have microcode and RISC philosophy dictates a careful evaluation of the cost and benefit of special hardware.

The emerging approach is to use small amounts of special hardware to detect "unusual" conditions and then trap to a handler for those conditions [Hill 87,Sun 87,Taylor 86,Ungar 86]. This seems to be a very effective approach; Taylor predicts SPUR Lisp performance to approach that of specialized Lisp machines. However, this burdens the trap handling architecture of existing systems beyond their design limits.

This article will show that current trap architectures are too slow to be effective for dynamic language traps and proposes changes that make trapping much better than inline code. Section 2 discusses the particular requirements of Lisp trap handling. Section 3 talks about the limits of current architectures with examples drawn from SPARC and SPUR. Section 4 proposes some trap handling features and estimates their cost and benefit in the context of both SPARC and SPUR. Section 5 is a summary.

2 Requirements

The requirements for Lisp trap handling are largely an extension of those in existing systems. Lisp is still interested in having the system handle interrupts, page faults, fatal errors, etc. However, Lisp and other dynamic languages can benefit from additional trap features.

These features fall into two major areas: operand/result type checking and storage management. Type checking is used to ensure that the operands to an operation are compatible with the operation and that the results are appropriate (e.g. do not overflow). Storage management features allow a Lisp system to implement read and write barriers to support modern garbage collectors.

2.1 Type Checking

In its full generality, Common Lisp [Steele 84] defines generic arithmetic on eight distinct numeric types. Any combination of numeric operands may appear on any operation. If the operands are non-numeric, the proper error condition must be raised.

Lisp machines use microcode and multi-way branch hardware to implement macroinstructions with the required functionality. Lisp implementations on conventional hardware have used a variety of techniques including out-of-line generic arithmetic routines, compile time type inferencing, and type declarations. Tag checking on conventional hardware typically consumes 16% to 18% of total execution time [Shaw 88].

The SPARC and SPUR RISC architectures contain instructions for fixed length integer (fixnum) arithmetic that cause a trap to occur if the operands are not both integers or if an overflow occurs. SPARC has `TADDccTV` and `TSUBccTV` instructions that do the necessary checking for fixnum operand types and overflow results. Nearly all SPUR's arithmetic, logical, and compare operations trap for inappropriate operands or overflows.

This can be a very effective approach. Shaw notes that "fixnum arithmetic is the dominant type of arithmetic in the test programs". Fixnums are 28%–51% of all data accesses and 54%–92% of all objects allocated. The trapping instructions permit the most common generic operations to be performed in a single instruction and still have the full generality Lisp requires. 12%–34% of SPUR's dynamic instructions do tag checks while less than 1% of the instructions actually trapped [Taylor 86] [3].

When a trap occurs, the trap handler must emulate the trapping instruction or enter the appropriate error context. For emulation, the handler (with or without hardware assistance) must decode the trapping instruction, fetch the operands, perform any required data conversion, do the operation, and place the results in the instruction's destination. For errors, the trap handler must return control to the Lisp error handler with sufficient information about the error and sufficient state to allow the error handler to repair the error (perhaps with the aid of the user) and continue.

The trap handler must have a certain amount of support to emulate an instruction. Primarily, it must have access to the primitive Lisp environment. The handler needs to access the operands, allocate storage for converted operands and results, and return a result as if it had come from the trapping instruction. The access and allocation requirements imply that the trap handler can itself cause traps, either for virtual memory faults or

[3] The question of how to effectively tag data in RISC machines is not one this paper will address any further. It is worth noting that Lisp does not require a large number of tag types. Shaw showed that one tag bit (integer/pointer) identifies 31% of the dynamic data in his benchmarks, two bits identifies 67%–98%, while three bits identifies over 98%.

storage management traps.

2.2 Storage Management

Garbage collection is a critical issue for Lisp system performance. It must have minimal impact on interactive response, impose minimal overhead on program execution, and interact well with virtual memory systems. Garbage collection has been justly accused of violating the basic premises of virtual memory. While virtual memory assumes locality of reference, most garbage collectors exhibit little, if any, locality. They tend to sweep the entire virtual space, making only a few references to each object in the space.

A great deal of research has gone into this problem with commendable success. Lisp machines now support very effective garbage collection. Dynamic storage management is not only inexpensive [Moon 84], but actually improves locality of reference [Courts 88]. Lisp machines use considerable amounts of special hardware, particularly in the memory mapping, to support storage management.

Garbage collection algorithms for conventional systems have been less successful. Until recently, Lisps on conventional machines used simple stop-and-copy garbage collectors. These collectors require a pause in Lisp execution while the garbage collector traverses the entire Lisp heap finding live objects and copying them. This approach requires neither a read nor a write barrier. However, it imposes a delay in interactive response (often several minutes) and destroys the virtual memory working set.

More recently, several commercial Lisp implementations have begun using generational stop and copy collectors to reduce these problems. This approach requires only a write barrier which can be implemented with inline software. These collectors have been more effective, but limit the practical virtual memory size of the Lisp and require much more physical memory than systems that use both a read and a write barrier.

Steenkiste shows that writes to the heap constitute about 1.7% of the dynamic instructions executed. A software write barrier check will consist of about 12 instructions, 4–6 of which will be executed if no barrier fault is detected. This implies that a software write barrier adds about 7%–10% to the runtime, which is consistent with Moon's estimate of 10% and Shaw's 7%–14% for Lisp systems [Moon 84,Shaw 88]. This makes software write barriers expensive, but acceptable.

SPUR provides a hardware write barrier. Each data item has a two bit generation number as part of its tag. A trap is generated when an item with a youngar generation tag is stored through a pointer with a older generation tag.

The Explorer^{tm} associates generation with the address of an object rather than with its pointer. Each entry in the memory map has two bits indicating the generation number of that area of storage. When one pointer is stored through another, address translation is done on both. The memory system sets a microcode flag if a pointer to a young storage area is being stored into an old area [Greenblatt 83].

A read barrier is generally easier to implement in hardware than a write barrier, but more difficult in software. Reads from the heap (which require a read barrier check) are 4.3 times more common than writes [Steenkiste 87]. A software read barrier can be implemented with roughly the same number of instructions as a write barrier. Because of the increased frequency, it will have a run time penalty that is about 4 times greater than the write barrier or 28%–40%, unacceptable in most systems.

The hardware for the Explorer's read barrier is simple. The same memory map that contains the generation numbers also contains an oldspace bit. Pointers fetched from the heap are translated through the memory map and a microcode flag is set if the pointer references oldspace. [Ellis 88] shows that read and write barriers can be implemented using standard memory protection hardware. [Krueger 88] shows how RISC architectures can be extended to implement read and write barriers.

2.3 Performance

It's clear that not all dynamic languages are alike. For example, data for Smalltalk indicates that only 3.9% of the tagged stores trap[Ungar 86], while Taylor's Lisp data (for programs that did any generation traps) ranges from less than 1% to nearly 91%, with a typical number being around 13%. A designer of a Lisp system who based the trap architecture on the Smalltalk data will be somewhat disappointed with the Lisp performance.

The performance of trap handlers can be critical to overall system performance. While trapping is a good strategy to deal with less common events and data types, "less common" does not necessarily mean "rare". Unfortunately, there is little data on trap frequency. Taylor notes that 0%–0.64% of all instructions cause write barrier traps and 0%–0.89% cause tag faults depending on benchmark. (He also notes that 0%–19.4% of all instructions do generation checks while 13%–35% do tag checks.)

If we speculate that read barrier traps are proportional to write barrier traps, then an estimate for total trap rate might be:

$$\begin{aligned} &Trap\ rate = \\ &\quad tag\ traps + \\ &\quad write\ traps + \\ &\quad (read/write\ ratio * write\ traps) \end{aligned} \tag{1}$$

Using Taylor's numbers we get a trap rate of 0%–4.3%. An average trap handler length of 25 instructions would double the runtime of the worst case program.

Trap handler performance also competes against inline code. Ignoring some secondary effects such as cache performance, the formula for the trade-off is:

$$\begin{aligned} &Trap\ overhead \leq \\ &\quad \frac{check\ frequency}{trap\ frequency} * check\ code\ length \end{aligned} \tag{2}$$

Using Taylor's data for the Boyer benchmark's write barrier checks, the trap overhead should be less than $\frac{2.9\%}{0.16\%} * 5$—about 90 instructions. This does not include the actual work done in the trap handler since that needs to be done in either case.

It is easy to get fooled by averages. There is a wide distribution of check and trap frequencies. A number of Taylor's benchmarks (9 of 17) did checks, but did not trap. For those programs, traps improve performance regardless of trap cost because the inline checks are eliminated. Taylor's worst case program (fft) trapped on 91% of the write barrier checks and 4.6% of the tag checks.

2.4 Requirements Summary

Trap architectures for Lisp systems have requirements that are significantly different from more conventional systems. The most important requirement is that the handler must be able to emulate some operation that is not directly supported by the hardware. Trap handlers need interfaces with garbage collection and Lisp level error handlers. This means the trap handler must have ready access to the Lisp environment.

Furthermore, the trap handler cannot accept some restrictions that are often placed on trap handlers. For example, a Lisp trap handler must reference arbitrary objects in the Lisp environment, which means it must be able to tolerate page faults i.e., the trap handler must also be able to trap.

Because Lisp traps occur at a much greater frequency than traditional system traps, the trap architecture must minimize the runtime (software) effort to get to the trap handler, determine what must be done, and do it.

Programs that cause unusually high numbers of traps will suffer greatly with slow trap handlers. Unfortunately, there are interesting programs (such as floating point intensive programs) that exhibit high trap rates. Without some care in the design of the trap architecture, the system will not be usable by those programs. It's worth spending some hardware to speed trap handling.

3 Limitations of Current Architectures

Current trap architectures are not designed for handling the traps of a dynamic language. They are intended to handle errors (e.g. address faults) or conditions requiring large amounts of processing (e.g. page faults). These trap architectures are inadequate for dynamic languages for three key reasons. First, all traps enter kernel mode. The user has little or no control over how the trap is handled. Second, there is insufficient support for rapid trap execution. Third, trap handlers cannot tolerate traps themselves.

3.1 Kernel Mode Trap Handlers

Most existing trap architectures (including SPARC and SPUR) expect the kernel to handle all traps[4]. If all traps enter the operating system, either the trap handlers for dynamic languages are built into the kernel or the kernel needs mechanisms to return control to the user for certain traps.

Fundamentally, the kernel is the wrong implementation level for dynamic language trap handlers. The kernel is both over privileged and under privileged. It's over privileged in the sense that it can generally do things users can't, such as accessing protected memory. If the trap handler is emulating a user instruction, it needs to do so with the the same access rights as the

[4]The SPARC processor enters supervisor state for all traps, while SPUR stays in user state for some traps. Both systems vector the traps through a table that cannot be modified by the user. Both schemes effectively constrain the kernel to fielding all traps.

user program, otherwise security holes may be opened. The kernel is under privileged because trap handlers usually run at low levels within the kernel and often do not have access to higher level functions such as file systems or communications.

Different dynamic languages need different trap handlers. They have different encoding schemes, different data types, different semantics, and different garbage collectors. The kernel needs a different set of handlers for each implementation of each language and must switch handlers with each process switch.

On a more mundane level, kernel trap handlers cause problems unless all the dynamic languages and the kernel are written and maintained by the same vendor. Kernel vendors are understandably reluctant to put "strange" code into the privileged portion of the system. Even then, the details of coordinating kernel releases with all the dynamic language releases seem overwhelming.

SPUR and the Sprite Operating System [Ousterhout 87] provide a much more flexible mechanism. A user program can "register" trap handlers with the kernel that will be used when a particular trap occurs. These trap handlers are part of the user process and run in user mode. The kernel calls the registered handler after a trap occurs, passing information such as the decoded trapping instruction and its operands. When returning, the trap handler can return results as if it came from the trapping instruction or it can cause the instruction to be re-executed.

This approach eliminates most of the functional difficulties of kernel resident trap handlers. Unfortunately, it does nothing to deal with the performance problems. In fact, the additional interface requirements slightly aggravate the situation.

3.2 SPUR Trap Architecture

Let's examine what SPUR does for the tag trap caused when instruction operands are not both fixed length integers. The numbers in parenthesis are the approximate[5] number of instructions required to perform the operation.

1. Preserve trap state and re-enable traps (35)
2. Decode trapping instruction and recover operands (63)

[5]The counts are approximate because there is some variance in path length and because some instructions contribute to more than one operation.

3. Find user handler and enter it (31)
4. User handler returns(via trap). Preserve state and re-enable traps. (24)
5. Place results in destination register and resume with next instruction. (35)

This is a total of 188 instructions for the trap overhead. With this overhead, a floating point operation will be about 200 times slower than the equivalent small integer (fixnum) operation.

The code is all hand coded assembly language. In a few places, the code is written more for generality than speed. However, the majority of the work is required by the trap architecture and is not "waste". About half of it (90 instructions, items 1, 3, and 4), is caused by having to enter and exit the kernel twice for each trap. Most of the remainder (items 2 and 5) is present because the software must emulate the hardware instruction fetch, decode, operand fetch, and result write.

3.3 SPARC Trap Architecture

While not available in any current commercial systems, the same approach to dynamic language traps could be used for the SPARC. The instruction counts for a tag trap handler interface are:

1. Preserve trap state and enter user mode (17)
2. Decode trapping instruction and recover operands (45)
3. Find user handler and enter it (21)
4. User handler returns (via trap). Place results in destination register and resume with next instruction. (23)

SPARC requires a total of 106 instructions for the interface. While better than SPUR, it is still unacceptable. Decoding the instruction is simpler because the SPARC has full 32 bit left and right shifts while SPUR can only shift 3 bits left and one bit right.

Recovering the operands and returning the results is far easier because SPARC's `SAVE` and `RESTORE` instructions can transfer data directly from one register window to another. On the other hand, the trap handler must correctly set SPARC's condition codes for the emulated instruction.

3.4 Limitations Summary

Both SPARC and SPUR have serious limitations on traps for dynamic languages. It is difficult for a user to get control after a trap occurs. The handler must use software to decode the trapping instruction and returning an emulated result is awkward. All this degrades performance to such an extent that inline code is faster for most programs.

4 Features

This section proposes a few additional features needed for good performance on dynamic language traps. Those features have very low implementation cost and yield much improved performance. The purpose is to give the user program control over certain traps, eliminate undue restrictions on code within trap handlers, provide information about the trap to the handler, and allow the handler to return a result as if it had come from the trapping instruction.

The features are first described in general terms that apply to many architectures. Then specific modifications to SPUR and SPARC are proposed with an analysis of cost and benefit.

4.1 User Trap Control

It's clear that the user program is the most effective place to handle certain traps, while the operating system must handle others. Traps should be divided into two groups: user and system. The system traps include:

- reset
- error
- interrupt
- page fault and protection violation
- window overflow and underflow
- Illegal or privileged instruction
- Half of the trap codes for a trap instruction.

which are basically the traditional traps.

The user traps include:

- tag trap

- overflow
- GC trap (read and write barrier faults)
- Unaligned address trap.
- Half of the trap codes for a trap instruction.

which are basically the traps added for dynamic language support.

System traps vector through a table in the kernel space[6]. User traps vector through a table in the user's address space, indicated by a special register, the User Trap Base Register (UTBR), which can be modified by the user.

System traps have priority over user traps. If both occur during an instruction, the system trap will be taken and the user trap ignored. Re-execution of the instruction after the system trap will generate the user trap if the trap condition is still present.

A user trap is best viewed as a forced subroutine call. The current register window pointer is incremented (a system window overflow trap may occur), the PC and NextPC[7] are stored in the new window, user traps are disabled, and execution continues at the proper location in the user trap vector. The processor remains in user state. In addition, information about the trapping instruction and it's operands are written to user accessible special registers.

System traps use the system window and do not generate a window overflow trap. They vector through the system trap table, disable system traps, and put the processor into supervisor state. The trap instruction information is *not* written to the special registers. This allows system traps to occur transparently, even in user trap handlers.

4.2 Trap Information

A large part of the tag trap handlers is devoted to recovering information (opcode and operands) that the hardware had available at the time of the trap. A major improvement in trap performance can be achieved by making the information directly accessible to the trap handler. What's needed are four special registers that can be read by user programs.

[6]SPUR has the table at a fixed location in low memory while SPARC uses a special register, the TBR, to hold a pointer to the trap vector.

[7]Most RISC machines (including SPUR and SPARC use delayed control transfer instructions. This means the next instruction to be executed may not be related to the trapping instruction. Two program counterss are a necessary part of the trap state.

The registers are:

`Opcode` — Contains the opcode of the trapping instruction.

`Op1` — Contains the *value* of the first source operand of the trapping instruction.

`Op2` — Contains the *value* of the second source operand of the trapping instruction.

`Dest` — Contains the register number of the destination register of the trapping instruction.

These registers are loaded with the appropriate values as part of a user trap. System traps do *not* modify them. If they were modified by a system trap, system traps could not be tolerated in user trap handlers. While a parallel set of registers would be useful for system traps, they probably do not "carry their weight" due to the lower frequency of system traps.

Each user trap handler should copy these special registers to general registers in the handler's window. At that point, user traps may be re-enabled. In fact, the only real purpose of disabling user traps is make sure the trap information registers are preserved until they can be read by the trap handler[8].

4.3 Handler Return

A user trap handler does one of two things on return. If the handler has corrected whatever condition caused the trap, the trapping instruction will be re-executed as if the trap had not occurred. If the handler has emulated the trapping instruction, a value must be returned as if it had come from the trapping instruction and execution continued with the next instruction. The former is what system trap handlers do with traps like virtual memory faults. Nearly all trap architectures provide good support for that style of trap return.

[8] An alternative implementation is to have the hardware trap logic place the values directly in general registers in the new window—this is already done for the PC and NextPC. This simplifies the trap handler code and eliminates the need to disable user traps. However, it makes the trap logic considerably more complex. Either the register file needs more write ports or the trap logic needs to force instructions into the pipe that copy the necessary values into the register file.

Architectures do not provide good support for instruction emulation by a trap handler. What needs to be done is for the emulated result to be placed in the destination register of the trapping instruction and any condition codes to be set appropriately.

Results are normally set in the register window *before* the window being used by the trap handler. The SPUR requires 35 instructions to do this and it must be done in kernel mode to prevent interrupts or other traps from destroying the trap handler window during the (brief) time it is running in the trapping instruction's window. This requires 24 more instructions to enter the kernel. The problem is less difficult on SPARC, which is able to use the RESTORE instruction to return a result directly.

4.4 SPUR Implementation

SPUR needs three modifications to implement the proposed trap architecture. It should partition it's trap vector into user and system vectors, it needs to add the user readable special registers, and it needs to add an instruction to return an emulated result.

The instruction VALUE_RETURN takes two arguments, the value to be returned and the number of the register to return it to (both arguments in registers). The instruction reads the return value and the destination register number out of the current register window, moves the window pointer to the previous window, and stores the return value in the proper destination register. It can be implemented with minor extensions to the existing SPUR data paths.

The 59 instructions SPUR needs to return an emulated result are reduced to 2:

```
/*  The trap handler is now executing in the register window
immediately below the trapping instruction's window.  Register values
are:
NEXT_PC_REG   -- the address of the next instruction to be executed.
DEST_REG      -- the number of the destination register of the
                 trapping instruction.
RESULT_REG    -- the value to be returned.
*/
/* Go back to user program */
jump_reg        NEXT_PC_REG
value_return    DEST_REG,RESULT_REG
/*In delay slot, return value */
```

The JUMP_REG instruction continues the user program at the next instruction after the trapping instruction. The VALUE_RETURN executes in the delay slot of the jump and returns the emulated result and restores the current window pointer.

4.5 SPUR Performance

The suggested changes have a major impact on trap performance. The 188 instructions from the SPUR's tag trap handler are reduced to 9:

```
/*Get trap information */

rd_special opcode,OPCODE_REG
rd_special op1,OP1_REG
rd_special op2,OP2_REG
rd_special dest,DEST_REG

/* Enable user traps*/
rd_special upsw,TMP_REG
or TMP_REG,TMP_REG,UTRAP_ENABLE
wr_special upsw,TMP_REG

/* Begin trap handler */

     ⋮

/* Go back to user program */
jump_reg NEXT_PC_REG
value_return DEST_REG,RESULT_REG
```

The reduction in instruction count is pretty spectacular. However, it's important only as it contributes to performance at the user level. As mentioned in section 2.3, there are two ways to measure it. Against inline code, we can rewrite equation 2 to show the "equivalent inline instructions" of the trap overhead:

$$\begin{aligned} &Equivalent\ inline\ instructions = \\ &trap\ overhead * \frac{trap\ frequency}{check\ frequency} \end{aligned} \tag{3}$$

For the Boyer benchmark's write barrier checks, the trap is cheaper than an inline check of $9 * \frac{0.16\%}{2.9\%}$ instructions—about a half an instruction. Since inline checks seem to cost about 4–6 instructions, the new trap architecture is a major improvement. The old architecture has $188 * \frac{0.16\%}{2.9\%}$ or about 10 "equivalent inline instructions" which indicates most checks are better done inline.

The actual execution time cost of the trap overhead is proportional to the percentage of instructions that trap. Taylor shows that tag traps occur on 0%–0.89% of the instructions executed. The cost of tag checking with a nine instruction trap overhead will be 0%–8%. Compare this to Steenkiste's 24% and Shaw's 16%–18% cost for inline tag checking.

Taylor also shows generation (write barrier) traps occur on 0%–0.64% of the instructions executed for an overhead of 0%–6%. Shaw estimates inline write barrier checks to cost 7%–14%. Using traps for both tag and write barrier checks will give a net performance improvement in the 9%–32% range.

4.6 SPUR Hardware

Little additional hardware is needed to implement this trap architecture on SPUR. The user trap vector requires an additional register (the UTBR) to hold the base address of the vector. This would be implemented in parallel to the existing Trap PC register used for the system traps. When a trap occurs, the proper register is selected based on trap type.

A user trap enable bit needs to be added to the existing User Processor Status Word register (UPSW). This controls whether user traps are taken or not. It also controls the updating of the user trap registers. If the bit is not set, the user trap registers are not changed.

The user trap registers are extensions to the existing SPUR special registers. `Op1` and `Op2` are placed in front of the ALU inputs and are loaded during the execute pipe stage. The `OpCode` and `Dest` registers are slightly more difficult. This information is not readily available after the instruction fetch stage. So two sets of temporary latches need to be added to carry the data through intermediate stages to the execute phase (when user traps occur).

Finally, the `VALUE_RETURN` instruction requires a fourth input be added to the multiplexor that selects the register to be modified. This allows the register number to be a data value rather than an immediate in the instruction.

None of these changes have an effect on the CPU's critical path. They also have only a minor impact on the area and gate counts. The trap information registers and their control logic require about 1300 transistors. The user trap base register and it's logic adds another 400 transistors. The other changes are trivial. The additional input to the multiplexor costs 10 transistors and adds 0.3% to the length of one data bus. The additional opcode requires only slight changes to the opcode PLA. Since the present SPUR CPU uses about 120,000 transistors, the proposed changes add about 1.4% to the chip.

4.7 SPARC Implementation

SPARC's changes are very similar to SPUR's. It should partition its trap vector into user and system vectors, add the instruction decode registers, and allow user programs to directly modify the integer condition codes. It is not necessary to add a `VALUE_RETURN` instruction. The `RESTORE` instruction can be used to perform the same function, although slightly more trap handler code is required.

SPARC has unique opcodes for reading and writing each special register. `RDPSR` and `WRPSR` read and write the processor status register while `RDTBR` and `WRTBR` read and write the trap base register. Following this model, ten new instructions need to added to read and write the five new registers[9].

A user trap enable bit needs to be defined. Unlike SPUR, SPARC does not have a user writable control register. It does have unused space in the processor status register (bits 19:14). It seems convenient to use one of those bits. By extending the privileged instructions `RDPSR` and `WRPSR` to allow the user to read and write only the condition codes and user trap enable, the user can control traps and easily emulate instructions that set condition codes.

4.8 SPARC Performance

The 108 instructions from SPARC's tag trap handler are reduced to 14:

```
/*Get trap information */
```

[9]This probably requires the least additional hardware, but it consumes considerable opcode space. An alternative is to borrow a couple of instructions from SPUR, `RD_SPECIAL` and `WR_SPECIAL`. These instructions use an immediate operand to specify the special register to be read or written.

```
rd %opcode,%OPCODE_REG
rd %op1,%OP1_REG
rd %op2,%OP2_REG
rd %dest,%DEST_REG
rd %psr,%PSR_REG

/* Enable user traps*/
or %PSR_REG,UTRAP_ENABLE,%TEMP_REG
wr %TEMP_REG,%psr

/* Begin trap handler */

     ⋮

/* Go back to user program */
        /* get current pc */
L1 call L2
sll %DEST_REG,2,%DEST_REG
L2 add %o7,%DEST_REG,%o7
jmpl     [%o7+(table-L1)],%0
wr       %PSR_REG,%psr

     ⋮

table
jmpl     %NEXT_PC_REG,%0
restore %DEST_REG,0,%0
jmpl     %NEXT_PC_REG,%0
restore %DEST_REG,0,%1

     ⋮
```

The first part of the handler picks up the trap information, enables user traps, and enters the handler. It requires seven instructions. The return section uses the destination register number as an index into a table of `jmpl-restore` pairs that return the emulated result, restore the window, and continue execution at the next instruction. This takes an additional seven instructions[10] for a total overhead of 14 instructions.

Translating the 14 instruction overhead through equation 3, shows that the equivalent inline instructions for the Boyer benchmark will be: 14 $*$

[10] A **VALUE_RETURN** instruction could reduce this from seven to three, which would speed up a program with a high trap rate by several percent.

$\frac{0.16\%}{2.9\%}$ or about .77 instructions. The unmodified architecture has an inline equivalent of about six instructions, indicating unmodified traps are about as efficient as inline code.

Tag traps will cost 0%–12% with the proposed changes and generation traps will cost 0%–9%.

4.9 SPARC Hardware

The hardware cost of making these changes on SPARC will vary depending on the implementation technology. The 20,000 transistor limit of the original gate array implementation makes it hard to justify the 1700 transistors needed. On the current custom VLSI implementations, the modifications should be no more expensive than they are for SPUR and are well worth doing.

5 Summary

Small amounts of hardware support can significantly improve the support for dynamic languages on RISC architectures. The technique of detecting "unusual" conditions and trapping is an efficient means of handling the wide range of operand types and storage management required by Lisp and other dynamic languages.

An architecture that wishes to support dynamic languages well must provide slightly more powerful trap mechanisms than previously necessary. In particular, the trap architecture must give the user control over certain trap types, the ability to rapidly recover information about the trap, and a means of returning emulated results. These features can be implemented as simple extensions to the trap architectures of most RISC systems and do not require large amounts of chip area or high degrees of complexity. They do yield substantial performance improvement over either inline software checks or existing trap architectures.

The same extensions that benefit dynamic languages can benefit conventional languages to a lesser extent. This trap structure makes it easy to have language specific handlers for integer overflow and similar conditions. Even C can benefit—Sun 4 C has a compiler option ("-misaligned") that allows access to integers that are not on a four byte boundary. It's implemented by using a subroutine instead of a `ld` instruction. With the proposed trap architecture, it could be a `ld` that traps if the data is misaligned.

A final word of advice: any trap handlers of importance must be fully written before the processor architecture is finalized. Without a complete implementation, it is much too easy to ignore some awkward feature that seriously damages performance.

5.1 Acknowledgments

I'd like to thank Bob Courts, Randy Katz, Steve Krueger, and Dave Patterson for their encouragement and thoughts in writing this paper. Shing-Ip Kong provided the details of the SPUR hardware implementation and needs special thanks. Finally, I'd like to thank everyone on the SPUR project who made me feel like a welcome part of the project for the last year.

References

[Baker78] Henry Baker, "List Processing in Real Time on a Serial Computer", *Communications of the ACM* 21(4), pp. 280-294, 1978.

[Bosshart 87] Patrick Bosshart et al., "A 533K-Transistor Lisp Processor Chip," *Digest 1987 International Solid-State Circuits Conference*, February 1987, IEEE, New York, pp. 203-203.

[Courts 88] H.R. Courts, "Improving Locality of Reference in a Garbage-Collecting Memory Management System." *Communications of the ACM*, September 1988.

[Ellis 88] John R. Ellis, Kai Li, and Andrew W. Appel, "Real-time Concurrent Collection on Stock Multiprocessors" Digital Systems Research Center Research Report 25. February, 1988.

[Greenblatt 83] Richard Greenblatt, private communication, Dallas, Texas, June 1983.

[Hill 87] Mark Hill, et. al. "Design Decisions in SPUR: A VLSI Multiprocessor", *IEEE Computer*, November 1986, pp. 8-22.

[Krueger 88] Steven D. Krueger, "VLSI-Appropriate Garbage Collection Support". To be published.

[Lieberman 83] H. Lieberman and C. Hewitt, "A Real-Time Garbage Collector Based on the Lifetimes of Objects.", *Communications ACM*, June 1983, pp. 419-429.

[MIPS 86] MIPS Computer Systems, Inc. *Performance Brief, April 24, 1986*, April 1986.

[Moon 87] David Moon, "Symbolics Architecture," *IEEE Computer*, January 1987, pp. 43-52.

[Moon 84] David Moon, "Garbage Collection in a Large Lisp System." *ACM Symposium on Lisp and Functional Programming*, Austin, Texas, 1984, pp. 235-246.

[Ousterhout 87] John Ousterhout, Andrew Cherenson, Fred Douglis, Michael Nelson, and Brent Welch, "The Sprite Network Operating System" *Computer*, Feburary 1988, also appeared as UC Berkeley Report No. UCB/CSD 87/359 June 1987.

[Steele 84] Guy L. Steele, *Common Lisp, The Language*, Digital Press, 1984.

[Steenkiste 87] Peter Steenkiste and John Hennessy, "Tags and Type Checking in Lisp: Hardware and Software Approaches," *Proceedings Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM/IEEE, October 1987, pp. 50-59.

[Steenkiste 88] Peter Steenkiste and John Hennessy, "Lisp on a Reduced Instruction-Set Processor: Characterization and Optimization." *Computer*, July 1988, pp. 34-45.

[Shaw 88] Robert Shaw, *Empirical Analysis of a Lisp System*, PhD dissertation, Stanford University, February 1988. Also appeared as Stanford Technical Report CSL-TR-88-351.

[Sun 87] Sun Microsystems, Inc., *The SPARC(tm) Architecture Manual* Revision 50 of August 8, 1987.

[Taylor 86] G.S. Taylor, P.N. Hilfinger, J. Larus, et al. "Evaluation of the SPUR Lisp Architecture." *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ACM, Tokyo, June, 1986,pp. 444-452.

[Ungar 86] David Michael Ungar, *The Design and Evaluation of a High Performance Smalltalk System*, PhD dissertation, U.C. Berkeley, Feburary 1986, Also appeared as Berkeley Technical Report UCB/CSD 86/287.