Performance Benefits of Tail Recursion Removal in Procedural Languages

Mark W. Bailey
Nathan C. Weston

TR-2001-2

June 2001

Department of Computer Science
Hamilton College
Clinton, NY 13323

Performance Benefits of Tail Recursion Removal in Procedural Languages

Mark W. Bailey
Department of Computer Science
Hamilton College
198 College Hill Road
Clinton, NY 13323
mbailey@hamilton.edu

Nathan C. Weston
Department of Computer Science
Hamilton College
198 College Hill Road
Clinton, NY 13323
nweston@hamilton.edu

ABSTRACT

The removal of tail recursion is a well-known optimization that is often applied by hand. It has been widely implemented in compilers for functional languages, generally through the use of continuation-passing style.

This optimization has also been implemented in some commercial compilers for **C** and other procedural languages. However, its effects in procedural languages have not been adequately documented, nor does the literature contain a detailed description of this optimization.

This paper discusses the removal of tail recursion as it applies to procedural languages, and describes a specific implementation within a **C** compiler. It also documents the performance benefits of this optimization, and compares them with the benefits of removing tail recursion by hand, at the source level.

The results show that this optimization can produce significant benefits, and can sometimes be performed better by the compiler than by hand. They suggest that, due to its low cost and high benefit, the removal of tail recursion is a useful optimization in a compiler for a procedural language.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*optimization, retargetable compilers*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*control structures, procedures, functions and subroutines, recursion*.

General Terms

Performance, Languages, Measurement

Keywords

Tail recursion, **C**, procedural languages, compilers, optimization.

1. INTRODUCTION

Conventional wisdom states that recursion is slow, and costly in terms of stack space. While this is often true, there are many compiler transformations which can reduce the cost of recursive functions. The transformation of tail recursion into iteration is perhaps the best known of these. With this transformation, tail recursive functions are optimized into equivalent loops, which avoid the costs procedure calls.

This optimization is widely used in the compilation of functional programming languages, and has been implemented in some commercial compilers [4]. However, its effects on **C** and other procedural languages have never been studied, and many compilers for **C** do not implement this basic optimization. **C** programmers, in general, continue to avoid recursion in favor of iteration. However, iterative solutions for many algorithms are much less elegant than their recursive equivalents. Recursive code is often more concise, and easier to read and maintain, than iterative code. Furthermore, as this paper will show, it need not carry any significant performance cost. We believe the bad reputation of recursion is largely the result of the failure of compilers to implement transformations on recursive functions, such as the removal of tail recursion. We attempt to demonstrate that tail recursion removal by the compiler matches hand optimization in performance, and provides significant benefits in terms of elegance and programmer convenience.

In this paper we provide a general discussion of tail recursion removal in procedural languages, a topic which has largely been neglected in the existing literature. We present evidence to show the utility of tail recursion, and of this optimization in particular. We also describe an implementation of this optimization in an existing **C** compiler, and document the performance and benefits of the optimization. Our results show that tail recursion removal is cheap to implement, and produces large performance benefits. Optimization by the compiler can even be superior to painstaking hand optimization at the source level.

The following section gives basic definitions of terms used in this paper. Section 2. presents related work. Section 3. provides examples where tail recursive code is preferable to iterative equivalents, and explains the utility of this optimization. Section 4. gives a high-level overview of the optimization. Section 5. describes our implementation in detail. Section 6. examines interactions with other optimizations. Section 7. presents benchmarks and results.

2. RELATED WORK

The removal of tail recursion was originally developed by McCarthy in his work on LISP [15], and has continued to be studied and implemented primarily in compilers for functional languages. Many of these compilers use continuation-passing style [18, 13], which has the effect of automatically removal tail recursion.

Some effort has been made at compiling functional languages to stack-based code—in particular to **C**, which has the advantage of being highly portable. Hanson developed efficient methods of stack allocation for tail-recursive languages [11]. Others have used **C** as a target language, but **C**'s implicit use of the stack, and lack of first-class functions have forced implementors to compromise either on proper tail recursion [16], or performance [20].

More recently, Clinger [5] created a formal, implementation-independent definition of proper tail recursion as a space complexity class. It is impossible to meet this definition in existing, stack-based implementations of procedural languages such as **C**, and there is no published data on tail recursion removal in these languages. However, it is possible to remove tail recursion in some cases, and this optimization is implemented in some commercial compilers [4].

3. OVERVIEW

Our discussion of tail recursion will rely on the following definitions: A procedure call *F* is a *tail call* if its caller, *C* does not do any additional processing after *F*. *C* must not return any value other than that returned by *F* (although it may return nothing at all). Tail calls are significant because, at the time a tail call is made, *C* no longer needs any of the data on its activation record (with the exception of the return address), nor does control ever need to return to *C*. Therefore the activation record may be discarded or reused by *F*.

A call is *tail recursive* if it is a tail call and is directly recursive—the caller and callee are the same function. This is a narrower definition of tail recursion than is commonly used in the discussion of functional languages [5].

Functional languages make heavy use of tail recursion, and recursion in general, and most implementations ensure that tail recursion is handled efficiently. In general this means a constant use of space, and the avoidance of a long string of returns. See Clinger for a complete discussion of proper tail recursion [5]. In general, these implementations rely on heap allocation, garbage collection, and continuations, none of which are available in **C**.

However, in **C** it is possible to convert tail recursion into simple iteration. This optimization can be performed at the source level, and is often done by hand [4]. The reason for this is as follows: when a function makes a tail call, it will not itself be performing any additional processing. When the callee returns, the caller will immediately return as well, simply passing along the return value (if any) of the tail call. Therefore, the caller no longer needs the environment stored in its stack frame, except for the return address. In fact, control does not need to return to the caller at all, and we can replace the procedure call with a simple jump. Argument passing is replaced with assignment, and the existing stack frame and registers and overwritten and reused. **C** does not support jumps between functions, but if the function is tail recursive, we can jump within it instead.

This can be accomplished at the source level with labels and gotos, or more elegantly with loops. First we insert assignments where needed to simulate argument passing. Next we add a label at the beginning of the function body, and replace the tail-recursive call with a goto to that label. In most cases, the label and goto can be replaced by a while loop.

As an example, consider a recursive implementation of the `strchr()` library function. Figure 1 shows the original **C** source code.

For each parameter, we insert the appropriate assignment immediately before the function call. Then we replace the call with a goto, as shown in Figure 2

A careful analysis reveals that we can combine the goto and the first if into an equivalent while loop. We invert the condition and move the body of the if outside of the while loop. We can also remove the useless assignment `c = c`. Everything else between the label and the goto becomes part of the loop. Figure 3 shows the final result of the transformation.

```
char * strchr(char *s, char c) {
       if(!*s)
              return 0;
       if(c == *s)
              return s;
       else
              return strchr(s + 1, c);
}
```

Figure 1: Source for recursive strchr()

```
char * strchr(char *s, char c) {
top:
       if(!*s)
              return 0;
       if(c == *s)
              return s;
       else {
              s = s + 1;
              c = c;
              goto top;
       }
}
```

Figure 2: Transformed strchr()

The original function used stack space linear in the length of the input string. Each call required pushing a return address, as well as saved registers, onto the stack. Also, on return the function had to pop each stack frame in turn, and execute one return for each call. In the transformed version, space usage is constant, and the call and return sequence is executed only once.

While this optimization can be performed by hand, at the source level, it is often desirable to leave the source in a tail recursive

```
char * strchr(char *s, char c) {
       while(*s) {
              if(c == *s)
                     return s;
              else
                     s = s + 1;
       }
       return 0;
}
```

Figure 3: Tail recursion transformed into a loop

form, as we will explain in the next section. Automatic optimization by the compiler gives the programmer the freedom to use tail recursion without worrying about its effects on performance.

4. MOTIVATING EXAMPLES

The use of tail recursion, and recursion in general, can make life significantly easier for the programmer. Many problems are solved most readily with recursive algorithms, and recursive algorithms are often shorter and clearer than their iterative counterparts. While some functions can be converted to iteration at the source level fairly cleanly, in other cases this conversion is not intuitive or reduces the readability of the code. Take, for example, the code for a binary search tree traversal shown in Figure 4. The structure of this function is immediately clear to the reader, and closely matches a high-level description of the algorithm.

While this function has two recursive calls, only the second is a tail call. The tail call can be optimized into a loop, but the non-tail call must remain recursive (or be transformed into iteration through use of a stack, which is significantly more involved). The result is shown in Figure 5.

```
void inorder(struct bst *n) {
              if(n->left)
              inorder(n->left);
       printf("%d ", n->value);
       if(n->right)
              inorder(n->right);
}
```

Figure 4: Tail recursive traversal of a binary search tree

```
void inorder(struct bst *n) {
       while(n) {
              inorder(n->left);
              printf("%d ", n->value);
              n = n->right;
       }
}
```

Figure 5: Hand-optimized traversal

The traversal of the left subtree is still handled by a recursive call, while the traversal of the right subtree is handled by the assignment to `n`, and the loop structure. Using two different constructs to do essentially the same thing, within the space of one function, makes the code more difficult to read and understand. This is, of course, a very small example, operating on a canonical data structure, and therefore the problem is a minor one. However, in more complex code this sort of hand-optimization can become very confusing.

With the benefit of automatic tail-recursion removal, both the tail-recursive and hand-optimized, iterative version will compile to equivalent machine code. This optimization allows us to preserve elegance at the source level, without sacrificing efficiency in the compiled program.

The qualitative observation that recursive code is often clearer is backed up by some quantitative data. Benander, Benander, and Sang performed a study of debugging performance in undergraduates, using recursive and iterative solutions for the same problems [1]. They presented students with logically incorrect **C** programs for searching and copying linked lists, and asked to debug them by inspection, which they argue remains the primary method of debugging once the problem has been isolated to a small segment of code. They found that students were significantly more likely to correctly fix the bugs in the recursive programs than in the iterative program. Their study lends support to the claim that some programs are more elegantly expressed in a recursive form, even in **C**, where iteration is traditionally favored.

Additionally, opportunities for this optimization are often overlooked. Quicksort is tail recursive, for example, as are many tree operations, and these algorithms are generally described in their standard, recursive forms. Unless a programmer makes a specific effort at performance tuning, it is likely that these functions will remain unoptimized in most implementations.

Not only does this optimization free the programmer from worrying about the performance costs of tail recursive functions, it allows functions to run which otherwise would not run at all. In a tail recursive function, stack space grows in $O(n)$, where n is the depth of recursion. However, this optimization removes the procedure call and recycles the same stack frame, which ensures constant use of stack space. Programs which otherwise might have run out of stack space can, after optimization, execute to completion. While the abundance of memory on desktop PC and servers makes this a minor concern, in embedded environments, where memory usage is a significant constraint [14], this may prove to be a greater benefit.

Finally, this optimization is useful because **C** is sometimes used as an intermediate language by compilers for higher-level languages [12, 20]. Although these compilers can perform tail recursion removal on their own, and generate iterative **C** code, the presence of this optimization in **C** compilers would further reduce the complexity of writing high-level compilers, and save compiler designers from unnecessary duplication of effort in optimizing tail recursion. There is also the possibility that, because this optimization functions at the machine code level, it could detect and remove tail recursive calls which are not recognizable at the source level, but take on a tail recursive structure at the machine code level due to optimizations or code generation techniques.

While opportunities for this optimization may be rare in **C**, it is easy to implement and comes at no cost. It does not increase code size and may reduce it slightly, depending on the calling conven-

tions of the target machine. Performing the optimization takes very little time, and when it can be applied, it will always improve execution speed of the resulting code. Because of its low cost and ease of implementation, this optimization should be a standard component to any optimizing compiler for procedural languages.

5. IMPLEMENTATION

5.1 vpo

This optimization was added to *vpo*, a retargetable optimizer. *vpo* works with a variety of frontends for procedural languages and can optimize code for over ten platforms. The implementation used in this work used the *lcc* frontend and generated code for the SPARC [10, 9, 19]. It operates on register transfer lists (RTLs), a machine-independent representation of machine code [6]. The optimization code is machine independent, and *vpo* can be retargeted by creating a new machine description, and implementing a set of machine dependent support functions [2, 3].

vpo is designed to work with naive code generators, and perform all optimizations at the object code level. It performs most optimizations found in commercial compilers to clean up the code. The strategy of naive code generation is continued within *vpo* itself. Many optimizations generate inefficient code sequences and rely on later optimization passes to improve them.

5.2 Identifying and removing tail recursion

While discussing our implementation, we will use the `strchr()` function shown in Figure 1. Figure 6 shows the initial control-flow graph of the RTL code generated by the frontend.

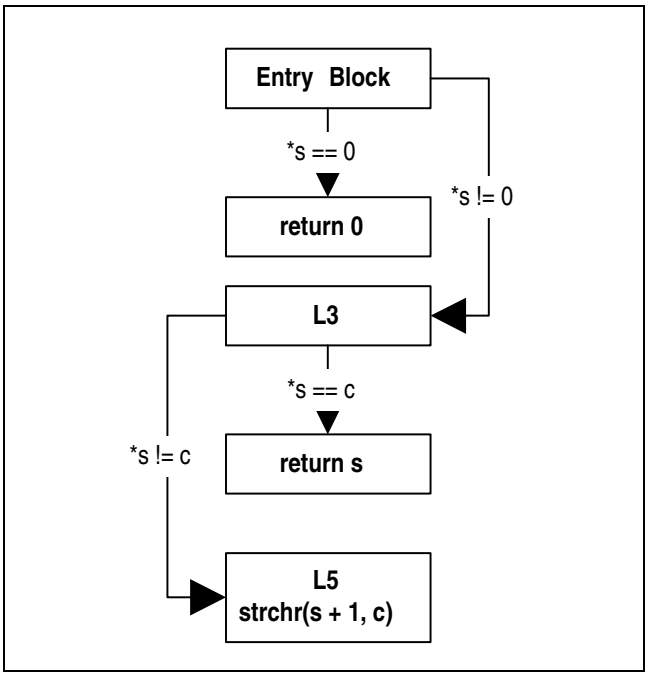

Figure 6: Initial Control-Flow Graph

To identify tail calls, we begin at the return block, and traverse the control-flow graph backward. Because control-flow optimizations may not been performed yet, some initial naive code sequences may still exist. The predecessors of the return block may be empty blocks, which simply fall through, or contain only a single, unconditional jump to the return block. In either case, we must then recursively examine the predecessors of those blocks. Our algorithm can follow these chains of branches indefinitely, although in practice we never encountered a chain longer than two jumps.

When we reach a block which contains code other than jumps, we begin at the end of the block and search backward until we find a procedure call. We then call a machine dependent helper function to identify whether that function is at the end of the block. This function takes into account the instructions which may need to occur after the procedure call in order to store a return value, and ensures that the callee's return value is not used in any way except to store it in the return register of the caller. We only search for procedure calls in blocks that are effectively predecessors of the return block (although they may follow a chain of jumps to get there), so if we identify that a procedure call occurs at the end of its block, we can be certain it is a tail call.

Once we have identified a tail call, we simply compare the names of the caller and callee. If they are the same, we have identified a tail recursive call, and we return a pointer to the RTL containing the call. The RTL has a pointer to the block which contains it, so we are also able to identify the block where the call occurs. Figure 7 shows the RTL which makes the recursive call in `strchr()`, along with the rest of its basic block.

```
//load s from a local variable
r[33]=R[r[30]+.p0.0_s;
//increment s, copy into arg register
r[8]=r[33]+1;
//copy c into argument register
r[9]=(B[r[30]+.p0.1_c]{24)}24;
//function call
ST=HI[strchr]+LO[strchr];
//copy the return value
R[r[30]+.10.0_1=r[33];
r[24]=R[r[30]+.10.0_1;
//return
PC=RT+8
```

Figure 7: Recursive call, before optimization

This RTL containing the call is then passed to `remove_tr()`. If this is the first invocation of `remove_tr()`, we must first insert a label at the top of the function being optimized. However, this label must be inserted after the function prologue code, or else the function will save registers multiple times. So, we scan through the top block of the function until we've passed the prologue code, then split the remainder of the block off to form the target block, which we label. The control-flow links are updated appropriately, so that the top block falls through into the target block, and the target block now has the predecessors which originally belonged to the top block. `remove_tr()` returns a pointer to the target block, which will be passed in if `remove_tr()` is called again.

Once we have a target block, we can replace the procedure call with a jump to it. We must set up the arguments of the function so that they will be modified appropriately on each iteration of the

loop. *vpo* initially models local variables, including arguments, as memory references. If the architecture passes arguments in registers, then the procedure prologue will copy them to the appropriate memory locations. Initially this can generate unnecessary memory references, but later optimization phases will attempt to assign these values to registers.

This naive code generation makes our work easier at this point. We can simply insert instructions before the procedure call to copy all parameters back to the stack. Figure 8 shows the call site after this step.

```
r[9]=(B[r[30]+.p0.1_c]{24)}24;
//store s and c back to  locals
R[r[30]+.p0.0_s]=r[8];
B[r[30]+.p0.1_c]=r[9];
//function call
ST=HI[strchr]+LO[strchr];
```

Figure 8: Call site after argument setup

We then replace the procedure call with an unconditional jump to the target block. Recall that the target block is immediately after the prologue code, which also ensures that all parameters are stored in memory. The effect is the same as if we had made a procedure call and allowed the prologue code to do the copying, except that we avoid the rest of the instructions in the prologue, such as saving and registers. Figure 9 shows the procedure call replaced with a jump.

```
//Argument setup
R[r[30]+.p0.0_s]=r[8];
B[r[30]+.p0.1_c]=r[9];
//Jump instead of a call
PC=L000
//End of block
```

Figure 9: Tail recursive call transformed into a jump

Finally, we clean up the control-flow graph to reflect our changes. A function may contain multiple tail calls—for example, binary search tree insert has one tail call to insert in the left subtree, and another to insert in the right subtree—so we continue our search. However `remove_tr()` modifies the control-flow graph, so we have to start our traversal from the beginning. We continue until our search completes without finding any tail recursive calls. Figure 10 shows the control-flow graph that results after applying this optimization.

This optimization generates even more unnecessary memory references, and makes no attempt to improve the structure of the loop it has created. However, later optimization passes will remove any unnecessary memory access (in many cases all memory references are removed, and arguments are simply stored in registers for the life of the function), perform control-flow optimizations, fill delay slots, and so on, resulting in highly efficient code. Because of the design of *vpo*, we do not need to worry about generating efficient code during our optimization, but can focus on simply getting code that works, and let the existing optimizations clean up after us. Figure 11 shows the final product. The loop structure has been changed to minimize branches, and the unconditional jump we inserted is now replaced with a conditional jump to a different target. All of the memory references are gone, and the character argument, which is constant, is not modified or copied at all.

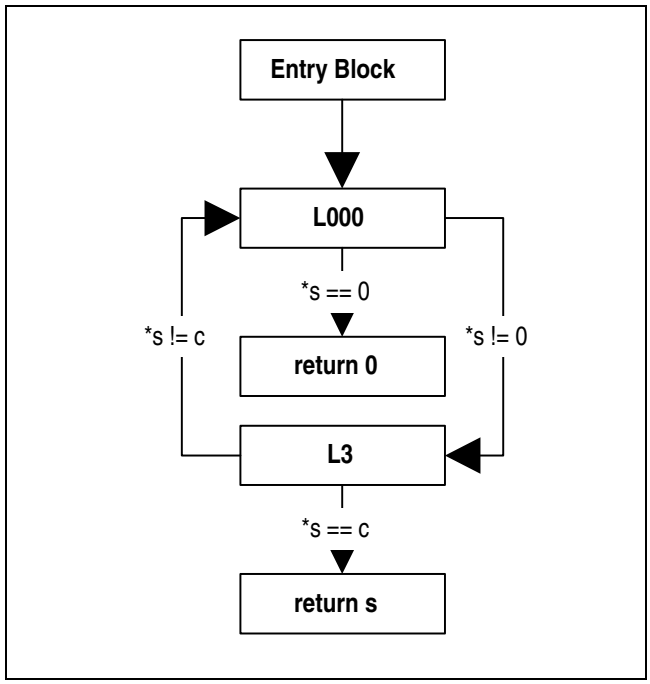

Figure 10: The resulting control-flow graph

```
//r[8] and r[9] hold c and *s
IC=r[8]?r[9];
//this used to be the function call
PC=IC!0,L5;
//argument passing has turned into a single
increment of s, placed in a delay slot
r[10]=r[10]+1;
```

Figure 11: Final Result

6. PHASE INTERACTIONS

Procedure calls present a significant barrier to optimization. The caller and callee may be compiled separately, making inter-procedural analysis impossible, and even if the callee is available, there is no guarantee that there will not be other callees which are compiled later. It is possible to analyze control-flow and detect loops across functions by optimizing at link time [17], but this has drawbacks as well. At this point, much of the information available from the source code is lost, and must be reconstructed or done without. This approach is therefore most suitable as an additional optimization pass, rather than a replacement for compile-time optimization, which means it increases compile times and adds significantly to the complexity of the system. Link-time optimization has not been widely adopted in production compilers.

Because our optimization removes procedure calls, it allows a significantly more thorough analysis to be performed on the trans-

formed function, and enables a number of optimizations which cannot otherwise be performed. In the following section we will examine interactions with common optimizations.

6.1 Loop Invariant Code Motion

Our optimization can produce significant gains when applied to loops which contain invariant code. The removal of the procedure call allows *vpo*'s loop invariant code motion routines to identify invariant code and move it outside the loop. The code identified was originally invariant across invocations of the procedure, rather than a loop, and therefore impossible to remove, but our optimization makes it visible to standard algorithms for code motion.

In the binary search tree traversal shown previously, the original version of the function loads the address of the format string (a constant) into an argument register during each invocation. This is necessary for two reasons: first, because the compiler cannot identify the loop, and second, because that argument register must be overwritten each time a recursive call is made. Once the call is removed, that argument register is no longer used to hold any value other than the address of the format string. The assignment is moved out of the loop, saving two instructions on each iteration.

6.2 Loop Unrolling

Loop unrolling is not available in the version of *vpo* used for this research, although it has been implemented in *vpo* [7]. While we were not able to test this empirically, in most cases we will probably not gain any additional benefit from loop unrolling. Most tail recursive functions are non-counting loops. Their execution count cannot be known in advance, even at execution-time, which greatly reduces the potential benefit of loop unrolling. *vpo*, and all of the compilers tested by Jinturkar, optimize counting loops only, and therefore would not have any effect on the majority of tail recursive functions.

However, it is possible for tail recursive functions to form counting loops. In some languages such as Scheme, where loop constructs are not commonly used, iteration is often expressed through tail recursive functions. If a compiler for such a language generated **C** code as an intermediate step, it might produce tail recursive counting loops. However, the practical benefits to be gained from loop unrolling in this situation remain unknown at this point.

6.3 Register Allocation

No significant interactions with register allocation were observed. All of our test functions are fairly small, which is characteristic of tail recursion functions, and therefore register pressure was minimal. Even without tail recursion optimization, our benchmarks did not access memory except when dereferencing pointer arguments, so there was no room to improve register allocation after our optimization was applied.

By removing a procedure call, our optimization should allow a more complete analysis of register usage, and better overall allocation. If arguments are passed in memory, this optimization has the potential to greatly reduce memory access by allowing data to stay in registers throughout the run of the function. On CISC architectures, this saves the cost of copying parameters into and out of memory, while on RISC architectures, it frees the register allocator from the need to funnel values into argument passing registers before a call. If there are multiple procedure calls, it will be necessary to swap different values into the argument passing registers for each. Once our optimization is performed, the values that were once arguments to the recursive call can now be freely allocated in any registers, which may reduce the amount of data movement which is necessary.

In practice, we did not observe significant interactions with register allocation. There are two factors which may explain this. Tail recursive functions tend to be fairly small, and do not require large numbers of registers. Also, our tests were performed on the SPARC, where registers are abundant. The combination of these two effects left little room for improvement in the original register allocations. It is possible that on a register-starved platform such as the x86, there would be more beneficial interactions, but this hypothesis has not been tested.

6.4 Control-Flow Optimizations

Tail recursive functions effectively form loops, but these loops are hidden from the compiler because they span across multiple invocations of the same function. This makes control-flow optimizations impossible in most cases, and at the very least prevents optimization of a significant portion of the function's structure. Explicitly converting tail recursion to an iterative form reveals the loop to the compiler, and allows normal optimization of control flow.

6.5 Inlining

Although inlining is not currently available in *vpo*, it has interesting potential interactions with tail recursion removal. Previous research has shown that inlining can be used to convert mutually recursive functions into a single, directly recursive function [12]. In some cases, the resulting procedure could be tail recursive, in which case it can be further optimized. So, the combination of inlining and tail recursion elimination has the potential to convert mutual recursion into iteration, which would no doubt result in significant performance gains.

6.6 Conclusion

Transforming a recursive call to iteration removes a major barrier to analysis and optimization, and enables many other optimizations. A tail recursive function, once transformed, can benefit from all of the optimizations that would apply to an equivalent iterative function. All of the performance costs associated with recursion are eliminated. In effect, our optimization allows us to perform a limited form of inter-procedural analysis at no additional cost, using existing algorithms. As we will see in the next section, this allows tail recursive programs to run as fast as, or faster than hand-optimized, iterative equivalents.

7. BENCHMARKS

We tested our optimization on several different benchmarks: insertion and traversal on binary search trees, quicksort, and the Ackerman recursive benchmark. In all cases, we tested a tail recursive version with and without our optimization, as well as a hand-optimized version in which recursion was transformed into iteration at the source level. Our tests consistently show a significant improvement from the removal of tail recursion, and show that our optimization produces code which runs as fast as iterative code.

7.1 Methods

We ran two benchmarks on each function: instrumentation with the *ease* system [8], and running-time measurements using the Solaris `times()` system call. Reported scores are for the function in question only, not the program as a whole. *ease* was used to obtain dynamic measurements of the number and type of instructions executed, memory references, and procedure calls. *ease* measurements reflect only instructions executed within the function itself, not in any functions it might have called.

The running-time measurements were taken by making five runs of the program, dropping the highest and lowest score, and averaging the remaining three. They reflect time spent in the function itself, as well as any functions it calls (such as quicksort's calls to partition()).

7.2 Binary Search Trees

This benchmark is a larger and more practical example. We implemented only insert and an in-order traversal (which increments the value each node). Insert is tail recursive in two cases, while traversal is tail recursive in one case, and has a second recursive call which is not a tail call. Both hand-optimization and *vpo* left the second recursive call intact, but transformed the tail call into iteration.

Our test program inserted integers at random (using a fixed seed for the random number generator to ensure that each run used the same sequence), then looked up all of those integers, as well as an equal number of random integers. A second test program built the tree similarly, then traversed it. Insertion was tested on a tree of 1,000 nodes, while traversal used a tree of 10,000.

7.3 Quicksort

We tested a standard, tail-recursive implementation of quicksort (and the equivalent hand-optimized version) on 10,000 integers, generated randomly with a fixed seed. The first element in the array was used as a pivot element in partitioning.

7.4 Ackerman Recursive Benchmark

The Ackerman benchmark is tail recursive in two cases, and has a third non-tail recursive call. In both hand and automatic optimization, the non-tail recursion was left intact, while the tail recursion was removed. The function was run once. It is worth noting that the Ackerman benchmark is non-trivial (although not difficult) to optimize by hand: this author encountered one minor bug in the process, and initially removed only one of the tail recursive calls.

Table 1: Dynamic Instruction Counts

	Number of Instructions			**% Improvement**	
	Tail Recursive	*Tail Recursive (optimized)*	*Hand Optimized*	*Tail Recursive (optimized)*	*Hand Optimized*
insert	194,002	130,576	136,499	48.6%	42.1%
traverse	133,458	120,513	129,171	10.7%	3.3%
quicksort	168,731	161,982	138,201	4.2%	22.1%
Ackerman	2,067,796	1,635,461	1,635,461	26.4%	26.4%

Table 2: Dynamic Memory References

	Number of Instructions			**% Improvement**	
	Tail Recursive	*Tail Recursive (optimized)*	*Hand Optimized*	*Tail Recursive (optimized)*	*Hand Optimized*
insert	417,082	62,794	62,794	564%	564%
traverse	314,291	176,211	310,028	78.4%	1.4%
quicksort	431,968	216,000	216,000	100.0%	100.0%
Ackerman	5,511,456	2,747,712	2,747,712	101.0%	101.0%

Table 3: Running Times

	Time in Seconds			**% Improvement**	
	Tail Recursive	*Tail Recursive (optimized)*	*Hand Optimized*	*Tail Recursive (optimized)*	*Hand Optimized*
insert	.294	.033	.034	791%	765%

Table 3: Running Times

	Time in Seconds			% Improvement	
	Tail Recursive	*Tail Recursive (optimized)*	*Hand Optimized*	*Tail Recursive (optimized)*	*Hand Optimized*
traverse	.061	.034	.039	79.4%	56.4%
quicksort	.284	.250	.250	13.6%	13.6%
Ackerman	2.183	2.122	2.122	2.9%	2.9%

Table 4: Dynamic Procedure calls

	Tail Recursive	**Tail Recursive (optimized)**
insert	11,736	1000
traverse	8,611	4,296
quicksort	13,499	6750
Ackerman	172,223	85,866

7.5 Results

The benefits of removing tail recursion, whether by hand or automatically, are clear. In some cases the running time was improved by over 500%, although generally the improvement was more modest. The degree of improvement is heavily dependent on the nature of the function being optimized. BST insert, which showed the largest improvement, is very densely tail recursive. Almost all of its processing is done through tail recursive calls. BST traversal, on the other hand, increments the value of each node as it traverses. This adds a load, a store, and an increment to the function body, which cannot be optimized away. Quicksort spends a large amount of time in the Partition function, which is naturally iterative, so the overall benefit of removing tail recursion is smaller. Finally, the Ackerman benchmark contains a non-tail recursive call (in addition to two tail recursive calls), which cannot be optimized and takes up a significant amount of processing time in all cases.

Compiler optimized code ran as fast or faster than hand-optimized code. On the whole, the performance differences between the two are very small. It appears that in some cases (such as BST insert) the compiler does a better job than hand-optimization, while in others (quicksort), hand-optimization is superior. There is evidence to suggest that this discrepancy is due to differences in the control flow between loops generated by the frontend, and those generated by our optimization.

The BST traversal benchmark stands out as the one case in which automatic optimization is the clear winner. It has a significantly lower instruction count, faster execution time, and far fewer memory references than the hand-optimized version. The *ease* measurements also reveal that the automatically optimized version makes fewer procedure calls: 4,296 compared to 8,612. A closer examination of the source code shows that the hand-optimized version actually descends one level deeper into the tree. The hand-optimized code checks whether the current node is null, then traverses the left and right subtrees. The tail recursive version, on the other hand, checks whether the subtrees are null before traversing them. It is possible to create an iterative version of this shallower traversal, which will run with speed comparable to the optimized tail recursive version. However, this requires complicating the function with a goto or while(1) loop. The current version of the benchmark was used because it represents the most natural, readable iterative solution.

Both hand and automatic optimization produce significant gains over unoptimized code, and unoptimized tail recursion should be avoided in frequently executed program segments. On the whole, automatic optimization produces code which is as good as that produced by hand optimization. In some cases it may be slightly better or worse, but in general the differences are minimal. Also, a significant benefit can be gained in cases such as BST traversal, where tail recursion facilitates a more efficient solution.

8. SUMMARY

Tail recursion is a powerful construct which unfortunately suffers from significant performance problems in most procedural language implementations. These problems can be greatly alleviated by incorporating tail recursion removal into the compiler, but this optimization has largely been ignored.

Tail recursion removal can be performed by hand, but it often results in more cryptic code, and is often overlooked. Automatic optimization, on the other hand, is easy to implement in a compiler and has little run-time cost. It will always identify opportunities for tail recursion removal, even in complex functions. It improves execution time without degrading the quality of source code, in some cases by more than 700%, and can even beat carefully hand-optimized code. Tail recursion removal frees the programmer to program in the most elegant and natural style, without worrying about performance problems resulting from inefficient language implementations.

9. REFERENCES

[1] BENANDER, A., BENANDER, B., AND SANG, J. An empirical analysis of debugging performance - differences between iterative and recursive constructs. *Journal of Systems and Software* (2000).

[2] BENITEZ, M. E., AND DAVIDSON, J. W. A portable global optimizer and linker. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation* (July 1988), pp. 329–338.

[3] BENITEZ, M. E., AND DAVIDSON, J. W. The advantages of machine-dependent global optimization. In *Proceedings of the 1994 Conference on Programming Languages and Systems Architectures* (Mar. 1994), pp. 105–124.

[4] BENTLEY, J. The cost of recursion. *Dr. Dobbs Journal* (1989), 111–114.

[5] CLINGER, W. D. Proper tail recursion and space efficieny. In *Proceedings of the ACM SIGPLAN '98 conference on Programming language design and implementation* (1998), pp. 174–185.

[6] DAVIDSON, J. W., AND FRASER, C. W. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems 6*, 4 (Oct. 1984), 505–526.

[7] DAVIDSON, J. W., AND JINTURKAR, S. Improving instruction-1evel parallelism by loop unrolling and dynamic memory disambiguation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture* (Nov. 1995), pp. 125–132.

[8] DAVIDSON, J. W., AND WHALLEY, D. B. Ease: An environment for architecture study and experimentation. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May 1990), pp. 259–260.

[9] FRASER, C., AND HANSON, D. *A Retargetable C Compiler: Design and Implementation*. Benjamin Cummings, 1995.

[10] FRASER, C. W., AND HANSON, D. R. A code generation interface for ANSI C. *Software–Practice and Experience 21*, 9 (1991).

[11] HANSON, C. Efficient stack allocation for tail-recursive languages. In *Proceedings of the 1990 ACM conference on LISP and functional programming* (1990), pp. 106–118.

[12] KASER, O., RAMAKRISHNAN, C., AND PAWAGI, S. On the conversion of indirect to direct recursion. *ACM Letters on Programming Languages and Systems* (1993), 151–164.

[13] KRANZ, D., KELSEY, R., RESS, J., HUDAK, P., AND PHILBIN, J. ORBIT: An optimizing compiler for Scheme. In *Proceedings of the SIGPLAN symposium on Compiler contruction* (1986), pp. 219–231.

[14] LIAO, S. *Code Generation and Optimization for Embedded Digital Signal Processors*. PhD thesis, MIT, 1996.

[15] MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 184-195 (1960).

[16] SERRANO, M. Bigloo reference manual. http:// kaolin.unice.fr/ serrano/bigloo/Doc/bigloo.ps.

[17] SRIVASTAVA, A., AND WALL, D. W. A practical system for intermodule code optimization at link time. *Journal of Programming Languages* (1993), 1–18.

[18] STEELE, GUY L., J. Rabbit: A compiler for Scheme. Tech. Rep. AITR-474, MIT, 1978.

[19] SUN MICROSYSTEMS CORPORATION. *The SPARC Architecture Manual, Version 7*, 1987.

[20] TARDITI, D., LEE, P., AND ACHARYA, A. No assembly required: Compiling Standard ML to C. *ACM Letters on Programming Languages and Systems* (1992), 161–177.

APPENDIX: SOURCE CODE FOR BENCHMARKS

BST Insert

Main Function

```
int main() {
    int i;
    struct bst *root;
    float total;
    long start, end/*, sum = 0*/;
    int values[NODES];

    srand(0);
    for(i = 0; i < NODES; i++)
       values[i] = rand();

    start = ttime();
    /*Insert random values into  the tree*/
    root = new_bst(values[0]);
    for(i = 1; i < NODES; i++) {
       insert(values[i], root);
    }

    end = ttime();
    total = (end - start) / 1000.0;
    printf("%3.3f ", total);
    return 0;
}
```

Tail Recursive

```
void insert(int value, struct bst *n) {
    if(value == n->value)
       return;
    if(value > n->value) {
       if(n->right)
           insert(value, n->right);
       else
           n->right = new_bst(value);
    }
    else {
       if(n->left)
           insert(value, n->left);
```

```
       else
           n->left = new_bst(value);
    }
    return;
}
```

Hand-optimized

```
void insert(int value, struct bst *n) {
    while(value != n->value) {
       if(value > n->value) {
           if(n->right)
              n = n->right;
           else {
              n->right = new_bst(value);
              return;
           }
       }
       else {
           if(n->left)
              n = n->left;
           else {
              n->left = new_bst(value);
              return;
           }
       }
    }
    return;
}
```

BST In-order Traversal

Main Function

```
int main() {
    int i;
    struct bst *root;
    int values[NODES];
    long start, end;
    float total;

    srand(0);
    /*Insert random values into the tree*/
    values[0] = rand();
    root = new_bst(values[0]);
    for(i = 0; i < NODES; i++) {
       values[i] = rand();
       insert(values[i], root);
    }

    /*Traverse the tree*/
    start = ttime();
    inorder(root);
    end = ttime();

    total = (end - start) / 1000.0;
    printf("%3.3f ", total);

    return 0;
}
```

Recursive

```
void inorder(struct bst *n) {
    if(n->left)
       inorder(n->left);
    n->value++;
    if(n->right)
       inorder(n->right);
}
```

Hand-Optimized

```
void inorder(struct bst *n) {
    while(n) {
       inorder(n->left);
       n->value++;
       n = n->right;
    }
}
```

Quicksort

Main Function

```
main()
{
   int i;
    int x[N];
    long start, end;
    float total;

    srand(100);
    for (i=0; i<N; i++)
       x[i] = rand();

    start = ttime();
    quicksort (x, N);
    end = ttime();

    total = (end - start) / 1000.0;
    printf("%3.3f ", total);

    return 0;
}
```

Tail Recursive

```
void quicksort(int A[], int n)
{
   int pivotIndex;

   if (n > 1) {
       pivotIndex = Partition(A, n);

       quicksort(A, pivotIndex);
       quicksort(&A[pivotIndex+1],
             (n-pivotIndex-1));
   }
}
```

Hand-Optimized

```
void quicksort(int A[], int n)
{
   int pivotIndex;

   while (n > 1) {
       pivotIndex = Partition(A, n);

       quicksort(A, pivotIndex);

       A = &A[pivotIndex + 1];
       n = n - pivotIndex - 1;
   }
}
```

Ackerman Recursive Benchmark

Main Function

```
int main() {
    long StartTime, StopTime, Result;
    float TotalTime;

    StartTime = ttime();
    Result = Ack(3,6);
    StopTime = ttime();
    TotalTime = (StopTime - StartTime)
        / 1000.0;

    printf("%d : %3.3f\n", Result,
        TotalTime);

    return 0;
}
```

Tail Recursive

```
int Ack(LowerBound, UpperBound) {
    if(LowerBound == 0)
       return UpperBound + 1;
    else if (UpperBound == 0)
       return Ack(LowerBound - 1, 1);
    else
       return Ack(LowerBound - 1,
              Ack(LowerBound, UpperBound
       - 1));
}
```

Hand-Optimized

```
int Ack(LowerBound, UpperBound) {
 top:
    if(LowerBound == 0)
       return UpperBound + 1;
    else if (UpperBound == 0) {
       LowerBound--;
       UpperBound = 1;
       goto top;
    }
    else {
       UpperBound = Ack(LowerBound,
         UpperBound - 1);
       LowerBound--;
       goto top;
    }
}
```