Stack Size Reduction of Recursive Programs

Stefan Schaeckeler
Department of Computer Engineering
Santa Clara University
500 El Camino Real
Santa Clara, CA 95053
sschaeckeler@scu.edu

Weijia Shang
Department of Computer Engineering
Santa Clara University
500 El Camino Real
Santa Clara, CA 95053
wshang@scu.edu

ABSTRACT

For memory constrained environments like embedded systems, optimization for program size is often as important, if not more important, as optimization for execution speed. Commonly, compilers try to reduce the code segment and neglect the stack segment, although the stack can significantly grow during the execution of recursive functions as a separate activation record is required for each recursive call. An activation record holds administrative data like the return address and the frame pointer but also the function's formal parameter list and local variables.

If a formal parameter or local variable is dead at all recursive calls, then it can be declared globally so that only one instance exists independent of the call depth. We found that in 70% of popular recursive algorithms and in all our real world benchmarks, it is possible to reduce the stack size by declaring formal parameters and local variables globally.

Architectures might impose a penalty in code size for accessing global data. On IA32, this stack size reduction starts to materialize for our benchmarks no later than in the fifth recursion.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Memory Management (garbage collection), Code Generation, Compilers, Optimization*; E.4 [**Coding And Information Theory**]: Data Compaction and Compression

General Terms

Languages, Performance, Experimentation

Keywords

Embedded systems, stack size reduction, recursion

1. INTRODUCTION

For memory constrained environments like embedded systems, optimization for size is often as important, if not more important than optimization for execution speed. The dominant factor in the costs of embedded systems is the total die size. As a large portion of the die is devoted to RAM and ROM, reducing die memory results in cheaper manufacturing costs. As smaller memories consume less power this translates also directly into longer running times of mobile devices. Optimizing the program for size may either lead to smaller dies, or alternatively to more functionality on the original die.

The programmer should be able to concentrate only on the correctness of the program and leave all optimizations to the compiler. Commonly, optimizing compilers try to reduce the code segment size and neglect the stack segment, although the stack can significantly grow during the execution of recursive functions as for each recursive call, a separate activation record is required. An activation record holds administrative data like the return address and the frame pointer but also the function's formal parameter list and local variables.

If a formal parameter or local variable is dead at all recursive calls, then it can be declared globally so that only one instance exists independent of the call depth. Embedded systems which make use of recursion can greatly benefit as the run-time memory consumption can be drastically reduced.

The rest of the paper is organized as follows. Section 2 gives a description and evaluation of our stack size reduction algorithm. Section 3 discusses an implementation and problems that might arise. Section 4 discusses experimental results for popular recursive algorithms as well as real world applications. Section 5 surveys related work in the field of code compaction and compression and section 6 concludes the paper.

2. STACK SIZE REDUCTION

This section gives a description and analysis of our stack size reduction algorithm.

2.1 Stack Size Reduction Algorithm

Declaring a formal parameter or local variable globally reduces the stack size of recursive functions, because only one instance will exist independent of the call depth.

A formal parameter or local variable that does not conflict with recursive calls can be declared globally, because the value of the variable before the call is not needed after the call. Hence the next incarnation of the function can reuse the space of the variable from the previous incarnation. As parameter passing is an implicit assignment of the actual

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'07, September 30–October 3, 2007, Salzburg, Austria.
Copyright 2007 ACM 978-1-59593-826-8/07/0009 ...$5.00.

parameter to the formal parameter, it must be explicitly modeled for globally declared formal parameters.

Example 1. Fig. 1 gives an example. The vertical bars in the figure represent live ranges for local variables `a` and `b`, and formal parameter `n`. In the original function, local variables `a` and `b` are both dead at the recursive call and can be moved into the global variable space. Local variable `a` is not needed after the recursive call at all, and although local variable `b` is needed, the original value before the call is not needed. Formal parameter `n` is alive at the recursive call and thus the value before the call is still needed after the call. This makes it necessary to allocate new space for `n` in each activation of `f`.

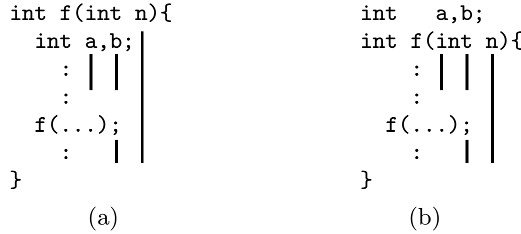

Figure 1: A Function before and after Optimization

Theorem 1 (Stack Size Reduction). *A formal parameter or local variable a_f in a recursive function f can be declared globally iff a_f is dead at all recursive calls of f.*

For the proof, we first need to define some notation. Notation $f \to g$ means function f calls function g, and $\to^*$ is the reflexive and transitive hull of $\to$. Let $\mathfrak{F}_g$ be the set of functions $\{h | g \to^* h\}$. Let $\mathfrak{p}(f)$ be the set of formal parameters and local variables of function f, and $\mathfrak{P}(\{f_1, f_2, \ldots, f_n\}) = \bigcup_{i=1..n} \mathfrak{p}(f_i)$. To unify the notation, let $\mathfrak{P}(f) = \mathfrak{p}(f)$, too. Let a_f be a formal parameter or local variable of function f, i.e. $a_f \in \mathfrak{P}(f)$.

Proof. If a_f is alive at a call $f \to g$, then this call introduces conflicts of a_f with each variable from $\mathfrak{P}(\mathfrak{F}_g)$.[1] If a_f is alive at calls $f \to g_i$, then these calls introduce conflicts of a_f with each variable from $\mathfrak{P}(\bigcup_i \mathfrak{F}_{g_i})$.

A call $f \to g$ is recursive if $g \to^* f$. If a_f is alive at a recursive call $f \to g$, then a_f conflicts with each variable from $\mathfrak{P}(\mathfrak{F}_g)$, which includes $\mathfrak{P}(f)$ of the following incarnation of f. Hence a_f from one incarnation of f cannot be coalesced with a_f of the following incarnation of f, and they must be declared locally.

Let g_i be the functions at those calls $f \to g_i$ a_f is dead. These calls cannot introduce additional conflicts of a_f with any variables from $\mathfrak{P}(\bigcup_i \mathfrak{F}_{g_i})$. This does not necessarily mean there are no conflicts of a_f with variables from $\mathfrak{P}(\bigcup_i \mathfrak{F}_{g_i})$ as they could be either introduced locally in f or by further calls $f \to h_j$ if $\mathfrak{P}(\bigcup_i \mathfrak{F}_{g_i}) \bigcap \mathfrak{P}(\bigcup_j \mathfrak{F}_{h_j}) \neq \emptyset$.

Again, a call $f \to g$ is recursive if $g \to^* f$. If a_f is dead at all recursive calls $f \to g_k$, then these calls don't introduce further conflicts of a_f with any variables from $\mathfrak{P}(\bigcup_k \mathfrak{F}_{g_k}) \subseteq \mathfrak{P}(f)$. If for other calls $f \to f_j$ ($\neq f \to g_i$) $a_f \notin \mathfrak{P}(\bigcup_j \mathfrak{F}_{f_j})$ (a_f can just be in $\mathfrak{P}(\bigcup_j \mathfrak{F}_{f_j})$ if it is alive at a call $f \to f_j$), then a_f from one incarnation of f does not conflict with a_f from the following incarnation of f. As variables that don't conflict can be coalesced, all a_f from every incarnation can be coalesced. This can be implemented by declaring a_f globally. □

[1] This call also introduces conflicts of a_f with each global variable alive in $\mathfrak{F}_g$, but these conflicts are irrelevant for this proof.

Corollary 1 (Tail-Recursion). *All formal parameters and all local variables of tail-recursive functions can be declared globally.*

Proof. A tail-recursive function is a function whose result either does not depend on a recursive call or is directly the result of a recursive call.

As the result of the recursive call is directly returned, no values are needed after the call. Hence all live ranges end before the call. □

2.2 Performance Analysis

The thin line in the time-space diagram of Fig. 2 shows a program in execution. The program starts with an initial amount of memory assigned to its process for code and global variables. During the execution of a non-recursive function from $t = 2$ to $t = 3$, space for the activation record is allocated and freed, again.

At time $t = 4$ a recursive function is called. For each successive call a new activation record is allocated. At time $t = 5$, the end of the recursion is reached and the function's call stack is unwound until the program continues executing the main function, again.

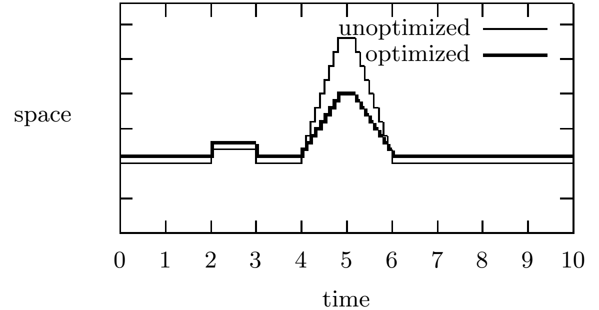

Figure 2: Time-Space Diagram of a Program

It might be possible to move some formal parameters and local variables into the global variable space yielding the thick line. During the execution of the main function and other non-recursive functions, more memory is needed because global variables exist for the whole execution of the program. During the first recursive call, equally much memory is needed, and during further recursive calls, significantly less memory is needed.

In other words, this optimization comes at the cost of requiring more memory during normal execution but chops off memory peaks during recursive calls.

Assuming variables are of word size, administrative data consist of two words for the return address and frame pointer, and assuming l of k formal parameters and local variables can be declared globally, then this optimization adds l words to the global variable space and saves, for a recursive call of depth n, $(n-1) \times l$ words, or $\frac{100 \times l}{2+k}\%$ (for large n) of stack space. The theoretical maximum saving is therefore $\frac{100 \times k}{2+k}\%$.

3. IMPLEMENTATION

The previous section discussed the algorithm in terms of a high level language. Often, compilers have to generate temporary variables but also try to keep variables in registers. To be able to consider all stack slots (formal parameters, local variables, and temporaries) for global allocation, an optimization pass should either work on the intermediate representation that already has all temporaries available, preferable after register allocation, or should be implemented as a post pass optimizer on assembler/machine code.

An optimization pass needs to keep track of live ranges of stack slots. If a live range for a stack slot does not conflict with the recursive calls, then this stack slot can be permanently allocated in the data segment, instead.

The following subsections discuss problems we encountered and observations we made while implementing our stack size optimization.

3.1 Stack and Data Segment Access

For our implementation we used the assembler output from the GNU tool chain for IA32. Gcc accesses data on the stack via the stack or frame pointer and a displacement. This makes the stack access fairly compact, e.g. on IA32 three bytes are needed for the movl displ <sp>, <reg> instruction and one byte for the push <reg> or pop <reg> instruction. On the other hand, global data is accessed immediate with the full 32 bit address, i.e. on IA32 six bytes are needed for the movl <address>, <reg> instruction.

Thus, for each access to global data a penalty of three or five bytes in code segment size must be paid.[2] This penalty is independent of the recursion depth. For an overall reduction in size, the recursion must be deep enough. If l stack slots á four bytes can be allocated in the data segment and they are accessed in the code segment v times, then there is a net gain starting between the $\lfloor \frac{3*v}{4*l} + 2 \rfloor^{th}$ and $\lfloor \frac{5*v}{4*l} + 2 \rfloor^{th}$ recursion. If some variables are accessed quite frequently and contribute much to the code size increase, then they can be kept on the stack. As we'll see in section 4, there is for our benchmarks a net gain no later than in the fifth recursion.

Beneficial for this code size optimization are architectures with special instructions for accessing the bottom of the address space. On such architectures the address field of load and store instructions can be of full width, but also of half or quarter width. With the first variant the full address space can be addressed and with the latter, shorter, variants only the bottom of the address space can be addressed. If the data segment is located at the bottom of the address space, then global data can be accessed with a lower size penalty.

Using a general purpose register as a base register for addressing the data segment with displacements increases the register pressure. Architectures with a *zero register* can use this register for addressing the data segment at the bottom of the address space with displacements quite compactly without increasing the register pressure.

3.2 Interaction between Code Size and Stack Size Optimization

During our study we observed that small optimizations for code size might be huge pessimizations for stack size. If several functions are called, then arguments are pushed on the stack before each call. To optimize for code size, gcc doesn't separately remove arguments after each call, but collectively after the last call. If the recursive call is not the first call, then the function goes into recursion with some unnecessary arguments on the stack.

Fig. 3 gives an example. Fig. 3a is pseudo assembler code often generated by gcc for size optimized code. Before the call to printf, its arguments a1 ... an are pushed on the stack. Similarly, before the call to f, its arguments b1 ... bm are pushed on the stack. Thus f goes into recursion not only with its own arguments on the stack but also with printf's. After the last call all arguments are removed by an adjustment to the stack pointer sp, although it would be more beneficial to remove printf's arguments from the stack before the recursive call to f (Fig. 3b).[3]

We suggest the removal before a recursive call of all arguments that have been pushed on the stack for previous function calls. That increases the code size by one instruction but saves for a recursive depth of n, n times the size of the removed arguments.

3.3 Caller-Save / Callee-Save Conventions

We observed also a problem arising out of caller-save / callee-save function call conventions. On IA32, the callers have to save registers EAX, ECX, EDX, FS, and GS on the stack if their values are needed after the call and the callee has to save registers EBX, ESI, EDI, EBP, DS, ES, and SS on the stack if they will be modified.

Such conventions are necessary for compiling functions separately. The callee doesn't know which registers are needed by the caller and the callers don't know which registers will be modified by the callee.

A recursive function is both a callee *and* the caller. If a callee-save register will be modified by a function then it must be pushed on the stack upon function entry. This must also be done for recursive functions even if the register is not alive at the recursive calls with the consequence of pushing the register in *each* recursive call.

In such instances, it might be worth putting recursive functions in wrapper functions so that for the recursive function, the calling convention can be broken. Only the wrapper needs to save the callee-save registers. Instead of saving them each time for every recursive call, the wrapper function needs to save them just once. On IA32, the overhead for the wrapper function is six bytes of code and four bytes on the stack.

We inspected the stack optimized assembler code of some popular recursive algorithms by hand and learned that some saved callee-save registers were dead at the recursive calls (see Table 1). Putting these functions into wrappers could have prevented these registers from getting pushed on the stack in each recursion.

4. EXPERIMENTAL RESULTS

For obtaining experimental results, we first compiled programs with size optimization enabled[4] and applied our stack size reduction algorithm on the emitted assembler code. Ta-

[2] Stack pointer manipulation operations for accessing the stack are not necessary for accessing the data segment and the penalty might be slightly less.

[3] On IA32, the stack grows downwards and the displacement needs not to be subtracted but to be added.

[4] To keep the recursion, we compiled tail-recursive programs with the additional flag -fno-optimize-sibling-calls.

```
f(...){
  push a1..an
  call printf

  push b1..bm
  call f
  sp = sp + sizeof(a1..an b1..bm)
}
```

(a)

```
f(...){
  push a1..an
  call printf
  sp = sp + sizeof(a1..an)
  push b1..bm
  call f
  sp = sp + sizeof(b1..bm)
}
```

(b)

Figure 3: Code Size Optimization resulting in Stack Size Pessimization

Table 1: Number of Callee-Save Registers

program-name	tail-rec.	number of callee-save registers	
		total number	dead at rec. call
divide		0	0
remainder	√	0	0
fac		1	0
fib		2	0
fib_fast		2	0
exp_2		0	0
exp		0	0
exp_fast		2	1
gcd	√	1	1
hanoi		3	0

ble 2 and 3 show the number of slots (á four bytes) per activation record without the return address and frame pointer as they always must be present.

4.1 Popular Recursive Algorithms

We tried to identify activation record slots for global allocation in ten popular recursive algorithms. Table 2 shows the results. Columns 3 and 4 show the number of slots before and after stack optimization. Columns 5 and 6 show the achieved saving and the theoretical maximum ($l = k$).

For 70% of the functions, there are savings between 12.5% and 60.0%. Two of these functions can be optimized by 16.7% to 25.0% of their theoretical maximum saving (`fib_fast` and `exp_fast`). Five of these functions can be optimized to their theoretical maximum saving. That are the two tail-recursive functions (`remainder` and `gcd`), but also tree non-tail-recursive functions (`divide`, `exp_2`, and `exp`).

As discussed in section 3.1, architectures might impose a size penalty for accessing global data. For IA32 Table 2 shows in columns 7 and 8 the code size in bytes before and after stack size optimization, and in column 9 the recursion level when the optimization starts to materialize. This recursion level n is the minimum $n \in N$ satisfying equation (1). It can be seen that there is a net gain starting between the second to fifth recursion.

$$\begin{array}{c}\text{code size before optimization}\\ + \text{n} \times \text{activation record size before optimization}\\ > \\ \text{code size after optimization} + \text{increase in data segment size}\\ + \text{n} \times \text{activation record size after optimization}\end{array} \quad (1)$$

4.2 Real World Applications

We have also isolated recursive functions from real world applications. As not many programs have recursive functions, we selected some programs from two benchmarks: MediaBench [8] and gcc code-size benchmark environment [3]. We tried to identify activation record slots for global allocation from flex (function `mark_beginning_as_normal` in `nfa.c`), bzip2 (function `snocString` in `bzip2.c`), and pgp (function `trace_sig_chain` in `keymaint.c`). Table 3 shows the results.

For all functions the stack size can be optimized. Flex' recursive function can be optimized to its theoretical maximum saving, and the remaining two recursive functions can be optimized by 40.0% to 61.5% of their theoretical maximum savings. For IA32 Table 3 shows in column 7 and 8 code size in bytes before and after stack size optimization. Column 9 shows there is, like in section 4.1, a net gain starting between the second to fifth recursion.

5. RELATED WORK

Classical optimizations [1] target mostly on execution speed and often code size increases, but code size may also decrease for certain strength reductions, dead code elimination, unreachable code elimination, common sub-expression elimination, constant folding, hoisting of common statements from branches, etc.

Most modern research on size optimization focuses on code size. For procedural abstraction, identical code sequences are identified and abstracted into functions [2] [5]. For procedure compression, procedures are separately compressed. Upon invocation, a procedure is uncompressed into a *procedure cache* and executed [4]. For cache-line compression, the program lies compressed in memory: a cache miss fetches the compressed instructions and decompresses them on the fly [7] [10]. The instruction width can be reduced by implementing 16-bit subsets of 32-bit ISAs [6] [9].

There is some research on reducing stack size. Variables with non-overlapping live ranges can be assigned to the same stack slot. This can also be combined with code size optimization [11].

6. CONCLUSION

Research in the field of program size optimization concentrates on code size. We have shown in this paper that the stack of recursive functions gives also opportunities for program size optimization.

Formal parameters and local variables that are dead at recursive calls can be declared globally so that only one instance exists independent of the call depth. We found that in 70% of popular recursive algorithms and in all our real world benchmarks, it is possible to reduce the stack size by declaring formal parameters and local variables glob-

Table 2: Activation Record Size and Code Size of Popular Algorithms

program-name	tail-rec.	number of slots per activation record before opt.	after opt.	saving actual	saving maximum	code size before opt.	code size after opt.	net gain starting at recursion level
divide		2	0	50.0%	50.0%	32	46	3
remainder	√	2	0	50.0%	50.0%	28	36	3
fac		2	2	00.0%	50.0%	34	no optimization possible	
fib		4	4	00.0%	66.7%	45	no optimization possible	
fib_fast		6	5	12.5%	75.0%	65	68	2
exp_2		1	0	33.3%	33.3%	28	35	3
exp		3	0	60.0%	60.0%	37	41	2
exp_fast		4	3	16.7%	66.7%	77	92	5
gcd	√	3	0	60.0%	60.0%	40	48	2
hanoi		11	11	00.0%	84.6%	75	no optimization possible	

Table 3: Activation Record Size and Code Size of Real World Applications

program-name	tail-rec.	number of slots per activation record before opt.	after opt.	saving actual	saving maximum	code size before opt.	code size after opt.	net gain starting at recursion level
flex		2	0	50.0%	50.0%	103	115	3
bzip2		5	3	28.6%	71.4%	115	143	5
pgp		13	5	53.3%	86.7%	462	490	2

ally. The savings for 50% of the popular algorithms are the theoretical maximum savings, and for 20% the savings are between 16.7% and 25.0% of the maximum savings. The savings for 33% of the real world programs are the theoretical maximum savings, and for 67% the savings are between 40.0% and 61.5% of the theoretical maximum savings.

Architectures might impose a penalty in code size for accessing global data. On IA32, this stack size reduction starts to materialize for our benchmarks no later than in the fifth recursion. To reduce this penalty, we suggested the use of certain compact addressing modes.

Caller-save / callee-save function call conventions contribute to live ranges at recursive calls. We showed how occasionally such live ranges can be eliminated with wrapper functions.

If a recursive function call is not the first call, then small optimizations for code size can become huge pessimizations for stack size. We observed, often arguments are not removed directly after calls, but collectively after the last call so that the recursive call goes with all previous arguments into recursion. To reduce the stack size, we suggested the removal of unnecessary arguments before recursive calls.

7. REFERENCES

[1] A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison Wesley, 2007.

[2] K. D. Cooper and N. McIntosh. Enhanced code compression for embedded risc processors. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 139–149, New York, NY, USA, 1999. ACM Press.

[3] CSiBE. `www.inf.u-szeged.hu/csibe/`.

[4] S. Debray and W. Evans. Profile-guided code compression. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 95–105, New York, NY, USA, 2002. ACM Press.

[5] C. W. Fraser, E. W. Myers, and A. L. Wendt. Analyzing and compressing assembly code. In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 117–121, New York, NY, USA, 1984. ACM Press.

[6] A. Krishnaswamy and R. Gupta. Profile guided selection of arm and thumb instructions. In *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 56–64, New York, NY, USA, 2002. ACM Press.

[7] C. Lefurgy. *Efficient Execution of Compressed Programs*. PhD thesis, University of Michigan, 2000.

[8] MediaBench. `euler.slu.edu/~fritts/mediabench/`.

[9] D. Seal. *ARM Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[10] A. Wolfe and A. Chanin. Executing compressed programs on an embedded risc architecture. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 81–91, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[11] X. Zhuang, C. Lau, and S. Pande. Storage assignment optimizations through variable coalescence for embedded processors. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 220–231, New York, NY, USA, 2003. ACM Press.