Optimizing the Stack Size of Recursive Functions

Stefan Schaeckeler [a,*], Weijia Shang [a]

[a] Santa Clara University, Computer Engineering Department, 500 El Camino Real, Santa Clara, CA 95053, USA

Abstract

For memory constrained environments, optimization for program size is often as important as, if not more important than, optimization for execution speed. Commonly, compilers try to reduce the code segment but neglect the stack segment, although the stack can significantly grow during the execution of recursive functions because a separate activation record is required for each recursive call.

If a formal parameter or local variable is dead at all recursive calls, then it can be declared global so that only one instance exists independent of the recursion depth. We found that in 70% of our benchmark functions, it is possible to reduce the stack size by declaring formal parameters and local variables global. Often, live ranges of formal parameters and local variables can be split at recursive calls through program transformations. These splitting transformations allowed us to further optimize the stack size of all our benchmark functions. If all formal parameters and local variables can be declared global, then such functions may be transformable into iterations. This was possible for all such benchmark functions.

Key words: Program size optimization, Stack size reduction, Memory compaction, Recursion, Recursion removal, Program transformation, Post pass optimization

1. Introduction

For memory constrained environments, optimization for program size is often as important as, if not more important than, optimization for execution speed. Programmers should be able to concentrate only on the correctness of their programs and leave all optimizations to the compiler. Commonly, size optimizing compilers try to reduce the code segment size but neglect the stack segment, although the stack can significantly grow during the execution of recursive functions because for each recursive call, a separate activation record is required. An activation record holds not only 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 global so that only one instance exists independent of the recursion depth. If a formal parameter or local variable is alive at a recursive call and passed as an argument to the called function, then this function may be modified to return the passed value so that it can be assigned back to the formal parameter or local variable. This effectively splits the live range at the function call, allowing the formal parameter or local variable to be declared global, as well. If all formal parameters and local variables can be declared global, then such functions might be transformable into iterative functions through simulating administrative data on stack by storing the return address once along with the current recursion depth. Recursive programs can greatly benefit from these three optimizations as they reduce the run-time memory consumption.

The strength of this optimization framework is the three successively optimizing phases. Even if recursion cannot be completely removed in the last phase, the previous phases may still (partially) optimize the function.

The rest of the paper is organized as follows. Section 2 gives a description of our basic stack size reduction algorithm and Section 3 discusses enhancements through live range splitting. Section 4 discuses how linear

* Corresponding author.
Email addresses: sschaeck@engr.scu.edu (Stefan Schaeckeler), wshang@scu.edu (Weijia Shang).

recursive functions can be transformed into iterations. Section 5 discusses related work while Section 6 concludes the paper.

2. Basic Stack Size Reduction

This section gives a description, a quantitative analysis, some implementation hints, and experimental results of our basic stack size reduction algorithm as published in [1].

2.1. *Basic Stack Size Reduction Algorithm*

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

A formal parameter or local variable that does not conflict with recursive calls can be declared global, 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 parameter to the formal parameter, it must be explicitly modeled for globally declared formal parameters.

Example 1 *The vertical bars in Fig. 1 represent live ranges for local variables* `a` *and* `b`*, and formal parameter* `n`*. In the original function of Fig. 1a, 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 at all after the recursive call, 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`*.*

Fig. 1. A Function before and after Optimization

Theorem 2 (Stack Size Reduction) *A formal parameter or local variable a_f in a recursive function f can be declared global iff a_f is dead at all recursive calls of f.*

For the proof, we first need to define some notation. Notation $f \rightarrow g$ means function f calls function g, and $\rightarrow^*$ is the reflexive and transitive hull of $\rightarrow$. Let $\mathfrak{F}_g$ be the set of functions $\{h \mid g \rightarrow^* h\}$. 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 (Stack Size Reduction) If a_f is alive at a call $f \rightarrow g$, 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 $f \rightarrow g_i$, then these calls introduce conflicts of a_f with each variable from $\mathfrak{P}(\bigcup_i \mathfrak{F}_{g_i})$.

A call $f \rightarrow g$ is recursive if $g \rightarrow^* f$. If a_f is alive at a recursive call $f \rightarrow g$, 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 local.

[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.

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 do not 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 do not conflict can be coalesced, all a_f from every incarnation can be coalesced. This can be implemented by declaring a_f global. □

Corollary 3 (Tail-Recursion) All *formal parameters and* all *local variables of tail-recursive functions can be declared global.*

Proof (Tail-Recursion) 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. *Quantitative Analysis*

The thin line in the time-space diagram of Fig. 2 represents 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 3, space for the activation record is allocated and then 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.

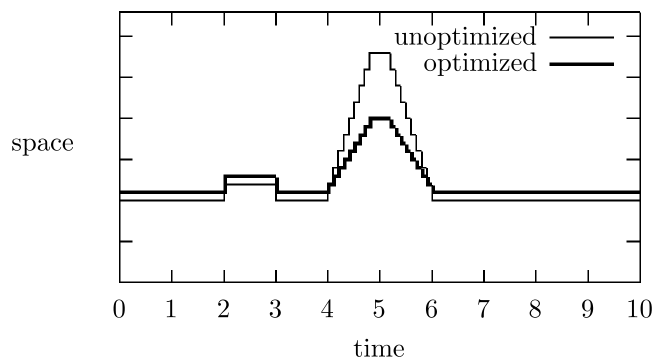

Fig. 2. Time-Space Diagram of a Program in Execution

It might be possible to declare some formal parameters and local variables global, 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, the same amount of memory is needed, but 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 truncates 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 k of m formal parameters and local variables are declared global, then this

optimization adds k words to the global variable space and saves, for a recursive call of depth n, $(n-1)k$ words, or $\frac{100k}{2+m}\%$ (for large n) of stack space. The theoretical maximum saving is therefore $\frac{100m}{2+m}\%$.

This optimization is not thread-safe and optimized functions may not be called from a thread. In a threaded context, a local variable exists per thread, but is shared among threads when declared global. On the other hand, modern thread implementations support *thread local storage* (TLS) for providing space for *thread local variables*. In the simplest case, such variables can be accessed relative to the thread pointer at a fixed offset [2, page 34].

2.3. *Implementation*

The previous subsections 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.

For our implementation we use the assembler output from the GNU tool chain for IA32. With six general purpose registers, the IA32 architecture has relatively few registers and the stack should be used more often. 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]

In a threaded context, the segment register `gs` holds on IA32 the address of the thread pointer and a general purpose register must be sacrificed to load it. With the thread pointer and a displacement, thread local variables can be accessed almost like local variables. The difference is that thread local variables lie *behind* the thread pointer and need to be accessed with negative displacements which are four bytes in size. The instructions are three bytes larger than their counterparts for addressing local variables and one byte larger than their counterparts for addressing global variables.

Assuming the l formal parameters and local variables $V = \{v_1, \ldots, v_l\}$ do not conflict with recursive calls, then any subset $S \subseteq V$ can be declared global. A function calculating the gain over the subset S and the recursion depth n can be given by *gain*: $2^V \times \mathbb{N} \mapsto \mathbb{N}$

$$gain(S,n) = (|c_{orig}| + n|a_{orig}|) - (|c_{opt}| + n|a_{opt}|) - (|a_{orig}| - |a_{opt}|) \tag{1}$$

$$= nI_a - (I_c + I_a) \tag{2}$$

where $|c_{orig}|$ and $|c_{opt}|$ are the code sizes of the original and optimized function, and $|a_{orig}|$ and $|a_{opt}|$ are the activation record sizes of the original and optimized function, and where the increase in code size is denoted by $I_c = -(|c_{orig}| - |c_{opt}|) > 0$ and the decrease in activation record size by $I_a = +(|a_{orig}| - |a_{opt}|) > 0$. I_a equals the increase in global variable space size.

The function *gain* is linear in n with a positive slope I_a and a negative y-intercept $-(I_c + I_a)$. Let

$$n_{gain} = \lfloor n+1 \mid gain(S,n) = 0 \rfloor = \lfloor (I_c + I_a)/I_a + 1 \rfloor. \tag{3}$$

For small recursion depths $n < n_{gain}$, the gain is negative or zero, but starting $n \geq n_{gain}$, the gain is positive. As we will see in Section 2.4, there is for our benchmarks a positive gain in no later than the fifth recursion.

Assuming variables are all of word size, and one word is four bytes, then I_a depends linearly on k by $I_a = 4k$. Furthermore, I_c depends non-linearly on S, because each of the k variables increases the code size

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

with a non-linear function i_c by $i_c(v_i) > 0$, where $i_c(v_i)$ is proportional to the number of accesses to variable v_i, hence $\sum_{i\in S} i_c(v_i) = I_c$.

Function *gain* can now be rewritten as

$$gain(S,n) = 4k(n-1) - \sum_{i\in S} i_c(v_i) \tag{4}$$

where n is the recursion depth and $S \subseteq V = \{v_1,\dots,v_l\}$ are the $k = |S|$ variables to be declared global.

The absolute value of the y-intercept and the slope are strictly monotone growing in k. Thus adding more variables to S increases both the absolute value of the y-intercept and the slope, but not necessarily n_{gain}. The absolute value of the y-intercept is strictly monotone growing in k because $i_c(v_i) > 0$ and the slope is strictly monotone growing in k because $\frac{\partial gain}{\partial k} = 4(n-1) > 0$ for $n > 1$. The largest slope is for $k = l$, e.g. for $S = V$, hence $\exists n_{max}\colon \forall S \neq V\colon \forall n \geq n_{max}\colon gain(S,n) < gain(V,n)$, e.g. starting n_{max}, the largest gain can be achieved by declaring all l variables of V global. We leave the calculation of n_{max} as an exercise and turn to a related problem.

If the recursion depth n is known by static analysis, profiling, or user assertion, then the optimal subset $S \subseteq V$ for a maximal gain can be calculated. If n is fixed, then *gain* becomes a function from 2^V to $\mathbb{N}$. A naïve algorithm would test all 2^l subsets of V for a maximum, but a thorough analysis can reduce the number of tests to k: *gain* can be broken into two terms, $4k(n-1)$ and $\sum_{i\in S} i_c(v_i)$. For *gain* to be large, the first term must be large and the second term must be small. They are competing requirements as the first term becomes larger by adding variables to S and the second term becomes smaller by removing variables from S.

For a fixed number $k = \hat{k}$ of elements in S, the first term becomes fixed, i.e. $4\hat{k}(n-1)$, hence the maximum depends solely on the second term. This term is minimal if the $\hat{k}$ smallest values from $\{i_c(v_i) \mid v_i \in V\}$ are added up. Let $g_{I_c,V}(i)$ be a function returning a unique variable $v \in V$ of the ith smallest value $i_c(v)$. Let $gain'$ be a function over the number of variables in V

$$gain'(k) = 4k(n-1) - \sum_{i\in\{1\dots k\}} i_c(g_{I_c,V}(i)) \tag{5}$$

Function $gain'$ can be calculated incrementally for $k = 2\dots l$:

$$\begin{aligned} gain'(1) &= 4(n-1) - i_c(g_{I_c,V}(1)) \\ &\vdots \\ gain'(k) &= gain'(k-1) + 4(k-1) - i_c(g_{I_c,V}(k)) \end{aligned} \tag{6}$$

Function $gain'$ is defined so that $max\{gain(S)\} = max\{gain'|S|\}$. It requires l calculations for finding a maximum for $gain'$. If $gain'$ has a maximum at $k = \hat{k}$, then *gain* has the same maximum at $S = \bigcup_{i\in\{1\dots\hat{k}\}} g_{I_c,V}(i)$, i.e. for the variables v_i with the $\hat{k}$ smallest values $i_c(v_i)$.

2.4. *Experimental Results*

To obtain experimental results, we first compiled our benchmark set consisting of 10 popular recursive algorithms with size optimization enabled[3] and applied our stack size reduction algorithm on the emitted assembler code. See [3] for results applying the algorithm directly on *C* code. Table 1 lists in columns 3 and 4 the number of slots per activation record before and after stack size optimization. The difference of both columns is the number of slots that can be declared global. The number of slots does not include the return address and frame pointer as they always must be present. Columns 5 and 6 show the achieved saving and the theoretical maximum ($k = m$).

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

For 70% of the functions, savings are between 12.5% and 60.0%. Three of these functions can be optimized by $16.7 - 33.3\%$ of their theoretical maximum saving (`fib_fast`, `exp_fast` and `exp`). Four of these functions can be optimized to their theoretical maximum saving. They are not only the two tail-recursive functions (`remainder` and `gcd`), but also two non-tail-recursive functions (`divide` and `exp_2`).

As discussed in Section 2.3, architectures might impose a size penalty for accessing global data. For IA32, Table 1 lists in columns 7 and 8 the code size in bytes before and after stack size optimization. The code size increases for the optimizable functions between 4.62% for `fib_fast` and 43.75% for `divide`. Column 9 shows the recursion level n_{gain} from Eq. (3) when the optimization starts to materialize. It can be seen that there is a net gain starting between the second to fifth recursion.

Table 1. Activation Record and Code Size

program-	tail-	activation record slots		saving		code size of function		net gain
name	rec.	before opt.	after opt.	actual	maximum	before opt.	after opt.	n_{gain}
`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	2	20.0%	60.0%	37	45	4
`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	

3. Live Range Splitting at Recursive Function Calls

As discussed in the previous section, formal parameters and local variables whose live ranges extend over recursive calls cannot be declared global. However, it might be possible to split live ranges at recursive calls, allowing more variables to be declared global [3].

First, the splitting algorithm is described. Then, some implementation hints are given and experimental results are presented.

3.1. *Splitting Algorithm*

Splitting is illustrated by an example, and then stated in a formal manner.

The transformation requires for intermediate steps a function returning two values, which can be modeled with structures. However, since *C* structures would cause the readability to suffer, we use an intuitive notation to assign and return two values.

Example 4 *The factorial function serves in Fig. 3 as an example. The standard implementation uses an implicit variable for holding the result of the recursive call. We made this explicit by introducing variable* `res`*. In this implementation (Fig. 3a), the live range of formal parameter* `n` *extends from the beginning of the function to the return statement. This implementation can be modified to return formal parameter* `n`*. The recursive call passes* `n-1` *to* `fac`*, which can be assigned back to* `n`*, and then the original value of* `n` *can be* rematerialized *by an increment of* `n` *(Fig. 3b). The first live range of* `n` *extends now from the beginning of the function to just before the recursive call as it is passed as an argument, and the second live range of* `n`

starts just after the recursive call as the recursive call is assigned to `n`*. Since* `n` *is dead at the recursive call, it can now be safely declared as a global variable. As stated in Section 2.1, parameter passing must be made explicit, i.e.* `...=fac(n-1)` $\rightarrow$ `n=n-1; ...=fac()` *(Fig. 3c). Assigning the return value of* `n` *to variable* `n`*, when coming back from recursion, is redundant and can be eliminated (Fig. 3d).*

```
int res;

int fac (int n){

  if (n>0){
    res=fac(n-1);

    return n*res;
  } else {
    return 1;
  }
}
```

(a)

```
int res;

{int, int} fac(int n){

  if (n>0){
    {res, n}=fac(n-1);
    n=n+1; /* remat. */
    return {n*res, n};
  } else {
    return {1, n};
   }
}
```

(b)

```
int res, n;

{int, int} fac(void){

  if(n>0){
    n=n-1; {res, n}=fac();
    n=n+1; /* remat. */
    return {n*res, n}
  } else {
    return {1, n};
  }
}
```

(c)

```
int res, n;

int fac(void){

  if(n>0){
    n=n-1; res=fac();
    n=n+1; /* remat. */
    return n*res;
  } else {
    return 1;
  }
}
```

(d)

Fig. 3. Live Range Splitting for Factorial

We need to define some notation next. Let $expr(v_1, \ldots, v_n)$ stand for an expression in variables $v_i, i \in \{1 \ldots n\}$ and let $expr_{v_i}^{-1}(v_1, \ldots, v_n)$ be the reverse expression with respect to variable v_i, i.e. $expr_{v_i}^{-1}(v_1, \ldots, v_{i-1}, expr(v_1, \ldots, v_n), v_{i+1}, \ldots, v_n) = v_i$. Let a store be a function σ: Var $\rightarrow$ Int, which maps all variables into integers. For the following discussion, let σ_{call} be the store before the function call statement, let σ_{f_create} be the store after the creation of the called function, and let σ_{f_term} be the store before the termination of the called function. Let $eval(expr, \sigma)$ be a function evaluating C expression $expr$ in store σ.

To split the live range of a formal parameter or local variable k at a recursive call, k must be passed as part of an expression to a formal parameter n, whose original value must be available or *rematerializable* at all exit points, i.e. $\exists p_i : \sigma_{f_create}(n) = eval(p_i, \sigma_{f_term})$. For rematerialization, values from any live range can be used alive at the rematerialization point. Live ranges can also be extended to the point of rematerialization provided the extension does not create an additional conflict with recursive function calls.

The function is modified to (a) return at all exit points the original value of n and (b) assign the result of the recursive call back to k. This modification changes the value of k. Hence, the value of k before the recursive call statement must be rematerialized afterward, again. Let k be passed as part of the expression $expr(k = v_1, \ldots, v_n)$ to the recursive function call. Since k holds after the function call statement the value $\sigma_{call}(expr(k = v_1, \ldots, v_n))$, k can be rematerialized by assigning $expr_k^{-1}(k = v_1, \ldots, v_n)$ to k if $v_i, i \in \{2, \ldots, n\}$ hold the values from before the call statement ($k = v_1$ usually does not). If the rematerialized expression is directly the variable k, then the rematerialization statement decays to $k = k$, and can be, as see in Example 4, eliminated.

Example 5 *Usually* $expr(k = v_1, \ldots, v_n)$ *is a simple expression like* $k - 1$*. For this expression, the inverse is* $k + 1$*, and thus* k *can be rematerialized by the assignment* $k = k + 1$*. For the more complex expression*

$sqrt(k) - v_2$, the inverse is $(k + v_2)^2$, and thus k can be rematerialized by the assignment $k = (k + v_2)^2$, provided v_2 still holds the value from before the call.

As the result of the recursive function call is assigned back to k, the assignment establishes the start of a new live range for k. The previous live range ends with passing k (as part of an expression) to the recursive function call. Hence, k is split at the recursive function call and can, if dead at all recursive function calls, be declared global.

With k as a global variable, the return statements can be explicitly modeled as assignments $k = p_i$. This eliminates the need for returning several arguments. As seen in the example, with $n = k = p_i$, the assignment statements $k = p_i$ decay to $n = n$ and can be completely eliminated.

3.2. *Implementation*

For splitting the live range of a variable k at a recursive function call, the reverse expression $expr_k^{-1}$ for a passed expression $expr(k = v_1, \ldots, v_n)$ must be rematerializable. This is the case if variables $v_2, \ldots, v_n$ do not change over the recursive call. This is clearly true for unaliased local variables. As most $v_2, \ldots, v_n$ are indeed unaliased local variables, we consider them, only.

A formal parameter n must be also available or rematerializable at all exit points. We observed that most formal parameters of recursive functions fall in one of these categories.

- A formal parameter is not changed at all. Hence, it can be returned trivially at all exit points.
- A formal parameter is decremented by a constant. The decrement can be undone at all exit points by a corresponding increment.
- If the formal parameter is decremented by a variable, instead, then the decrement can be undone if the variable has not changed (variables that change can be tried to be rematerialized, too).
- A formal parameter is a pointer to the stack, e.g. to a linked list. Practically, it is very difficult to rematerialize stack pointers.

We did not encounter many further parameter manipulations. Although rematerialization of formal parameters could be made arbitrarily complex or impossible, practically, it is often possible. We consider formal parameters that fall in the first three categories, only.

3.3. *Experimental Results*

For obtaining experimental results, we applied the splitting algorithm on assembler code output of Section 2. See [3] for results from applying the algorithm directly on *C* code. Table 2 lists in columns 3 and 4 the number of slots per activation record in the original implementation and after splitting. The difference of both columns is the number of slots that can be declared global. The number of slots does not include the return address and frame pointer as they always must be present. Columns 5 and 6 show the achieved saving and the theoretical maximum ($k = m$).

With splitting, the stack of all benchmarks, including the previously unoptimizable benchmarks, can be optimized by $23.1 - 66.7\%$. Benchmarks which cannot be optimized by the basic algorithm can be optimized with splitting by $23.1 - 66.7\%$ (`fac`, `fib` and `hanoi`). Benchmarks which can be partly optimized by the basic algorithm can be optimized further by $60.0 - 66.7\%$ (`fib_fast`, `exp` and `exp_fast`).

As discussed in Section 2.3, architectures might impose a size penalty for accessing global data. For IA32, Table 2 lists in columns 7 and 8 the code size in bytes before and after stack size optimization. The code size increases for those functions which are optimizable between 20.00% for `gcd` and 73.33% for `fib`. Column 9 shows the recursion level n_{gain} from Eq. (3) when the optimization starts to materialize. It can be seen that there is a net gain starting between the second to fourth recursion.

4. From Recursion to Iteration

With the previous algorithms from Sections 2 and 3, all temporaries, formal parameters, and local variables can be declared global for 80% of the benchmark functions. From there it is for many functions a small step

Table 2. Activation Record Size

program-name	tail-rec.	activation record slots: original	activation record slots: splitting	saving: actual	saving: maximum	code size of function: before opt.	code size of function: after opt.	net gain n_{gain}
divide		2	0	50.0%	50.0%	32	46	3
remainder	√	2	0	50.0%	50.0%	28	36	3
fac		2	0	50.0%	50.0%	34	50	4
fib		4	0	66.7%	66.7%	45	78	4
fib_fast		6	1	62.5%	75.0%	65	101	3
exp_2		1	0	33.3%	33.3%	28	35	3
exp		3	0	60.0%	60.0%	37	61	4
exp_fast		4	0	66.7%	66.7%	77	123	4
gcd	√	3	0	60.0%	60.0%	40	48	2
hanoi		11	8	23.1%	84.6%	75	99	4

towards the complete removal of recursion.

First, rules for transforming linear recursive functions into iterative functions are given. Then, some implementation hints are given and experimental results are presented.

4.1. *Iteration Transformation Rules*

For a (recursive) function call, temporaries, local variables, formal parameters, and administrative data are pushed on the stack. If the function has neither temporaries, nor formal parameters or local variables, then only administrative data—the frame pointer and the return address—remain on the stack. The frame pointer is redundant as the main usage is addressing stack objects. The return address is the address of the statement following the (recursive) call statement. If there is exactly *one* return address for recursion, then this address is pushed on the stack over and over again. This redundancy can be eliminated by storing the return address once along with the recursion depth, i.e. the number of occurrences. If there is only one recursive call statement, then there is only one return address, but several recursive call statements can have also identical return addresses as they might be the last statements within an if–then–else or case cascade. If several recursive call statements have different return addresses, but only one recursive call statement is actually executed per function invocation, e.g. the function is linearly recursive, then it might be possible to move the recursive call statement to a single point, e.g. a common post dominator point, and rematerialize the actual arguments, while keeping the semantic. Such transformed functions have then also only one return address for recursion.

The source level transformation rules given in Fig. 4 are for functions with exactly one recursive call statement. The assembler level transformation rules for such functions are very similar and not given here.

For the transformation, an integer variable is needed to keep track of the current recursion depth. The return address can be directly encoded in the label of the goto statement. On IA32, a function returns a result through register EAX. For a high level language transformation, this register must be explicitly exposed as a variable of the function's return type. We assume a return type of integer in Fig. 4.

Two global variables, `_depth` and `_result`, are declared for the transformation. The former variable must be initialized to zero upon function invocation. The recursive function call `x = f()` is rewritten as a jump to the beginning of the function. Before the jump, variable `_depth` is updated by an increment and after the jump, the return value is assigned to the proper variable `x`. A return statement is rewritten as a jump to the caller. Before the jump, variable `_depth` is updated by a decrement, and the return value is assigned to variable `_result` for the caller to pick up. If the recursion depth is zero, then the recursion is over and the function must return to the initial caller instead.

Global Declarations

```
int _depth;
int _result;
```

Function Prologue

```
int f(void){
  _depth=0;
  top:
```

Function Call Statement `x = f()`

```
_depth++;
goto top;
caller:
x=_result;
```

Return Statement `return z`

```
if (_depth==0){
  return z;
} else {
  _depth--;
  _result=z;
  goto caller;
}
```

Fig. 4. Rules for Transforming Recursion to Iteration

The result of the transformation is an iterative function. To make use of more compact addressing modes, all variables declared global in the transformation phases of Sections 2–4, can and should be declared local again. The function is then thread-safe again. For the discussions to follow we assume variables are declared local.

Example 6 *Fig. 5 shows the result of applying the iteration transformation on the factorial function of Fig. 3d. The resulting code gives many opportunities for further optimizations. We regard these optimizations as a separate problem.*

```
int fac(int n){

  int res, _result, _depth=0; top:

if(n>0){
  n=n-1;
  _depth++;
  goto top;
  caller:
  res=_result;
  n=n+1;
  if (_depth==0){
    return n*res;
  } else {
    _result=n*res;
    _depth--;
    goto caller;
  }
} else {
  if (_depth==0){
    return 1;
  } else {
    _result=1;
    _depth--;
    goto caller;
  }
}
```

Fig. 5. Iteration Transformation for Factorial

4.2. *Implementation*

As seen in Example 6, the iteration transformation of Fig. 5 results in inefficient code for a high level language transformation. This is also true for a transformation on assembler level. Hence, it is important to let conventional code optimization passes follow. For this reason, we performed the transformation phases of Sections 2–4 on *C* code. The results are iterative functions in *C* that can be compiled with gcc to size efficient assembler code.

We observed in Section 2.3 that the basic stack size reduction algorithm is imprecise when applied on *C* code. This is also true for the splitting algorithm. The imprecision is gone after the iteration transformation is applied, because the function is then iterative and no stack slots can conflict with any recursive calls.

4.3. *Experimental Results*

For obtaining experimental results, we applied the iteration transformation on *C* code and then compiled the resulting function to assembler. Table 3 lists the results for those functions having, after splitting, neither local variables nor formal parameters. [4] Columns 3 and 4 show the number of slots per activation record in the original implementation and after the iteration transformation. The number of slots does not include the return address and frame pointer as they always must be present.

Table 3. Activation Record Size and Code Size

program-name	tail-rec.	activation record slots		code size of function		net gain n_{gain}	iter-ative
		before opt.	after opt.	before opt.	after opt.		
`divide`		2	2	32	39	2	√
`remainder`	√	2	2	28	29	2	√
`fac`		2	1	34	44	2	√
`exp_2`		1	1	28	40	5	√
`exp`		3	2	37	45	2	√
`exp_fast`		4	2	77	73	1	√
`gcd`	√	3	3	40	54	3	√

All seven functions could be transformed into iterations. Six functions had just one recursive call statement and one, `exp_fast`, had two. Six functions can store all temporaries and local variables, including `_depth` and `_result`, in registers and only parameters are on the stack. Only `gcd` requires one temporary on the stack.

The equations used in this section for calculating *gain* and n_{gain} differ from the equations in Section 2.3, because the resulting functions do not go into recursion. The new equations for *gain* and n_{gain} are

$$gain(S,n) = (|c_{orig}| + n|a_{orig}|) - (|c_{opt}| + 1|a_{opt}|) \quad (7)$$

$$= n|a_{orig}| - (I_c + |a_{opt}|) \quad (8)$$

and

$$n_{gain} = \lfloor n+1 \mid gain(S,n) = 0 \rfloor = \lfloor (I_c + |a_{opt}|)/|a_{orig}| + 1 \rfloor. \quad (9)$$

For IA32, columns 5 and 6 show the code size in bytes for the original functions and for the optimized, iterative functions. The code size increases between −5.19% for `exp_fast` and 42.85% for `exp_2`. Column

[4] This does not include `fib`. Although it was possible to remove all stack slots on assembler level (see Table 2), it was not possible to remove all formal parameters and local variables on *C* code level. Apparently, some of gcc's optimizations are beneficial for live range splitting.

7 shows the recursion level n_{gain} when the optimization starts to materialize. Often the code size does not increase much in relation to the original activation record size, and often the number of activation record slots either remains the same or decreases. Hence, there is a net gain quite often as early as in the second iteration and for one function even in the first iteration, but also as late as in the third or fifth iteration.

5. Related Work

Classical optimizations [4] target mostly execution speed, often increasing code size, but code size may also decrease from certain strength reductions, dead code elimination, unreachable code elimination, common sub-expression elimination, constant folding, hoisting of common statements from branches, etc.

Modern research on program size optimization focuses on code size but neglects stack size, although the stack can significantly grow during the execution of recursive functions. The only significant works are on recursion removal.

Each recursive function can be transformed into an iterative function with the help of an explicit stack, but such a transformation does not reduce the size complexity and sophisticated transformations are required. We are not aware of much work related to the first and second phase of our stack size reduction algorithm (Sections 2 and 3). In this section, we discuss recursion removal algorithms and compare them to our stack size reduction algorithm, particularly to its last phase (Section 4).

Program schematology is the study of program behavior and equivalence based on schemas, i.e. based on abstract and uninterpreted programs. Paterson et al. show in [5] the equivalence of linear recursive schemas with iterative schemas by a *translation* which trades speed for size: any $O(n)$ linear recursive function can be computed by an iterative function in $O(n^2)$ time. Opposed to our first two phases, our last phase works only on linear recursive functions as well. Although it does not guarantee the successful removal of recursion, it guarantees, if successful, the same time complexity. By exploiting inverses like we did in the second phase, linear recursion can be transformed into iteration [6]. Likewise, by exploiting associativity and changing the order of evaluation, linear recursive functions can be transformed into iterations [6].

Burstall et al. studied in [7] transformations of functions, including recursion removal, based on the transformation rules definition, instantiation, (un)folding, abstraction, laws about primitives, and optionally redefinition, as well as on strategies for applying them. These strategies are somewhat ad-hoc, requiring intuition whereas our algorithm is systematic and programmable.

A further approach is based on incrementalization [8], e.g. on identifying an appropriate input increment operation and computing the function by repeatedly performing an incremental computation at the step of the increment [9]. Liu et al. have shown in [10] that incrementalization is very powerful and can optimize even the Ackermann function (with keeping some recursion).

6. Conclusion

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

The strength of our optimization framework is the three successively optimizing phases. Even if recursion cannot be completely removed in the last phase, the previous phases may still (partially) optimize the function.

In the first phase, formal parameters and local variables that are dead at recursive calls are declared global so that only one instance exists independent of the recursion depth. We found that in 70% of our benchmark functions, it is possible to reduce the stack size by declaring formal parameters and local variables global. The savings for 40% of our benchmarks are the theoretical maximum savings, and for 30% the savings are between 16.7% and 33.3% of the maximum savings. Architectures might impose a penalty in code size for accessing global data. On IA32, this optimization starts to materialize for our benchmarks no later than in the fifth recursion.

In the second phase, live ranges of formal parameters or local variables can often be split at recursive calls through program transformations. Splitting transformations allowed us to further optimize the stack

size. For *all* benchmark functions, including previously unoptimizable ones, we saved between 23.1% and 66.7% of stack space. On IA32, this optimization starts to materialize for our benchmarks no later than in the fourth recursion.

In the last phase, it was possible for all 70% of our benchmark functions without parameters and local variables to completely remove recursion. We believe we are the first to have investigated the impact of recursion removal on code size. Our iteration transformation drastically increases code size, but conventional optimizations can reduce code size again, resulting in a net gain starting the second to fifth iteration.

References

[1] S. Schaeckeler, W. Shang, Stack size reduction of recursive programs, in: CASES '07: Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, ACM Press, New York, NY, USA, 2007, pp. 48–52.

[2] U. Drepper, Elf handling for thread-local storage (2005). URL `people.redhat.com/drepper/tls.pdf`

[3] S. Schaeckeler, W. Shang, Live range splitting at recursive function calls, in: IIT '07: Proceedings of the International Conference on Innovations in Information Technology, 2007, pp. 337–341.

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

[5] M. S. Paterson, C. E. Hewitt, Comparative schematology, in: Record of the Project MAC conference on concurrent systems and parallel computation, ACM, New York, NY, USA, 1970, pp. 119–127.

[6] H. A. Partsch, Specification and transformation of programs: a formal approach to software development, Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[7] R. M. Burstall, J. Darlington, A transformation system for developing recursive programs, J. ACM 24 (1) (1977) 44–67.

[8] Y. A. Liu, T. Teitelbaum, Systematic derivation of incremental programs, Sci. Comput. Program. 24 (1) (1995) 1–39.

[9] Y. A. Liu, S. D. Stoller, From recursion to iteration: what are the optimizations?, SIGPLAN Not. 34 (11) (1999) 73–82.

[10] Y. A. Liu, S. D. Stoller, Optimizing Ackermann's function by incrementalization, in: PEPM '03: Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation, ACM, New York, NY, USA, 2003, pp. 85–91.

Stefan Schaeckeler received his M.S. degree in computer and information science from the University of Massachusetts, Dartmouth, MA, and the diplom degree in computer science from the Universität Stuttgart. He is currently a Ph.D. student in the Department of Computer Engineering, Santa Clara University. His main research interest is program compaction for embedded systems.

Weijia Shang received her B.S degree in electrical and computer engineering from Changsha Institute of Technology, Changsha, China in 1982 and the M.S. and Ph.D. degrees in electrical engineering from Purdue University, West Lafayette, IN, in 1984 and 1990, respectively. She is currently an Associate Professor in the Department of Computer Engineering, Santa Clara University. Her research interests include parallel processing, computer architecture, algorithm transformation, processor array programming, special purpose VLSI bit-level processor array design, and optimizing compiler technique.