

Live Range Splitting at Recursive Function Calls

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 size 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 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. This allowed us to optimize the stack size in 70% of our benchmarks.

Often, live ranges of parameters and local variables can be split at recursive calls through program transformations. Such program transformations allowed us to further optimize the stack size of all our benchmarks.

1. Introduction

For memory constrained environments like embedded systems, optimization for size is often as important, if not more important, as 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 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 reduced. If a formal parameter or local variable is alive at a recursive call and passed as an argument to that 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 globally, too. This results in a further reduction of the run-time memory consumption.

The paper is organized as follows. Section 2 describes our basic stack size reduction algorithm and section 3 discusses improvements through live range splitting. Section 4 surveys related work in the field of code compaction and compression. Section 5 discusses future work and section 6 concludes the paper.

2. Basic Stack Size Reduction

This section introduces our basic stack size reduction algorithm, evaluates it, and presents experimental results. For a more formal treatment we refer the reader to [8].

2.1. Basic Stack Size Reduction Algorithm

Declaring a formal parameter or local variable globally reduces the stack size of recursive functions, because only

```
int f(int n){
  int res;
  if (n==1) return 2;
  res = f(n-1);
  return res * res;
}
```

(a)

```
int res;
int f(int n){
  if (n==1) return 2;
  res = f(n-1);
  return res * res;
}
```

(b)

```
int n, res;
int f(void){
  if (n==1) return 2;
  n=n-1; res = f();
  return res * res;
}
```

(c)

Figure 2. Example function

```
int f(int n){
  int a,b;
  :
  :
  f(...);
  :
}
```

(a)

```
int a,b;
int f(int n){
  :
  :
  f(...);
  :
}
```

(b)

Figure 1. A function before and after optimization

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.

The vertical bars in Fig. 1 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*.

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 by assigning, before the call, the argument to the formal parameter. Variables in arguments to a (recursive) function call are *uses* before the call. The assignment, if there is one, of the return value of the function to a variable is a *definition* of this variable after the call. All other variables used on the right hand side of such an assignment are *uses* after the call. E.g., for $x=y * f(z)$, *z* is used before the call, *y* is used after the call, and *x* is defined after the call.

An example is given in Fig. 2 with function *f* calculating $2^{(2^{n-1})}$. Temporary variable *res* holds the result of

the recursive call, is then squared and returned. The live range of *res* is depicted in Fig. 2a and extends from after the function call to the end of the function. Hence, it can be declared globally (Fig. 2b). Formal parameter *n* can also be declared globally, because its live range, depicted in Fig. 2b, extends from the beginning of the function to just before the call. Because parameter passing is an implicit assignment, it's modeled explicitly, i.e. $res=f(n-1) \rightarrow n=n-1; res=f()$ (Fig. 2c).

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 other values are needed after the call, and all live ranges end before the call. Hence *all* formal parameters and *all* local values can be declared globally.

2.2. Quantitative Analysis

The thin line in the time-space diagram of Fig. 3 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.

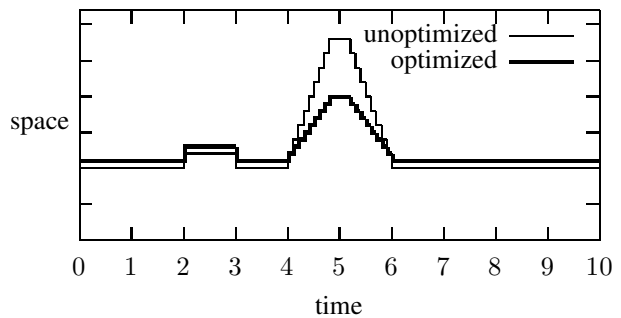


Figure 3. Time-space diagram

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, the same amount of 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 call depth of 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 $\frac{100 \times k}{2+k} \%$.

2.3. Experimental Results

We obtained experimental results for our benchmarks consisting of ten popular recursive algorithms. Table 1 shows in columns 3 and 4 the number of formal parameters and local variables in the original implementation and after basic stack size reduction. The difference of both columns is the number of formal parameters and local variables that can be declared globally. Columns 5 and 6 list the stack reduction saving and the theoretical maximum ($l = k$).

Table 1. Activation record size

program-name	tail-rec.	params. & local vars.		saving	
		original	basic	actual	max
divide		2	0	50.0%	50.0%
remainder	√	2	0	50.0%	50.0%
fac		1	1	00.0%	33.3%
fib		2	2	00.0%	50.0%
fib_fast		4	3	16.7%	66.7%
exp_2		1	0	33.3%	33.3%
exp		2	1	25.0%	50.0%
exp_fast		2	1	25.0%	50.0%
gcd	√	2	0	50.0%	50.0%
hanoi		4	4	00.0%	66.7%

For 70% of all benchmarks, the stack size can be optimized by 16.7% to 50.0%. Four of these benchmarks can be optimized to their theoretical maximum (divide, remainder, fib_fast, and gcd) and the remaining three benchmarks can be optimized by 25% to 50% of their theoretical maximum (fib_fast, exp, and exp_fast).

3. Live Range Splitting at Recursive Calls

As discussed in the previous section, formal parameters and local variables whose live ranges extend over recursive calls can't be declared globally. This section discusses a novel improvement to our basic algorithm: if a formal parameter or local variable is alive at a recursive call and passed as an argument to that 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 globally, too.

First, we illustrate splitting by an example. Then we state more formally the splitting conditions and the splitting algorithm. Finally, we present experimental results.

3.1. Example

The factorial function servers in Fig. 4 as an example in live range splitting. The splitting transformation requires for intermediate steps a function returning two values, which can be modeled with structures. Using C structures, the readability would suffer, though. Hence, we use an intuitive notation to assign and return two values.

In the original implementation (Fig. 4a), 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. 4b). The first live range of n extends now from the beginning of the function to just before the first 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, parameter passing must be made explicit, i.e. `...=fac(n-1) → n=n-1; ...=fac()` (Fig. 4c). Assigning the return value of n to variable n , when coming back from the recursion, is redundant and can be eliminated (Fig. 4d).

3.2. Live Range Splitting

Splitting Conditions. The conditions for splitting a live range of a variable v at a recursive call are: (A) The original value of a formal parameter n must be available or rematerializable at all exit points of the function. (B) A variable v , may it be a formal parameter, but not necessarily the same formal parameter n , or a local variable, must be passed directly or as part of an expression to that formal parameter

```

int res;

int fac (int n){

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

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

```

(a)

```

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;

{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 fac(void){

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

```

(d)

Figure 4. Live range splitting for factorial

n . If v is passed as part of an expression, then the original value before the call must be rematerializable after the call.

Splitting Algorithm. The function must be modified to return the original value of formal parameter n at all exit points, and the result of the recursive function calls at which v is passed to n and supposed to be split, must be assigned back to v . If v is passed as part of an expression, then the original value of v must be rematerialized afterward.

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

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

3.3. Experimental Results

We obtained experimental results for live range splitting. Table 2 shows in columns 3 and 4 the number of formal parameters and local variables in the original implementation and after splitting. The difference of both columns is the number of formal parameters and local variables that could be declared globally. Columns 5 and 6 list the stack reduction saving and the theoretical maximum ($l = k$).

With splitting, the stack of all benchmarks, including the previously unoptimizable benchmarks, can be optimized by 16.7% to 50.0%. Benchmarks which can't be optimized by the basic algorithm, can be optimized with splitting by 16.7% to 33.3% (`fac`, `fib`, and `hanoi`). Benchmarks which can be partly optimized by the basic algorithm, can be optimized further by 33.3% to 50.0% (`fib_fast`, `exp`, and `exp_fast`).

4 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

Table 2. Activation record size

program-name	tail-rec.	params. & local vars.		saving	
		original	splitting	actual	max
divide		2	0	50.0%	50.0%
remainder	√	2	0	50.0%	50.0%
fac		1	0	33.3%	33.3%
fib		2	1	25.0%	50.0%
fib_fast		4	2	33.3%	66.7%
exp_2		1	0	33.3%	33.3%
exp		2	0	50.0%	50.0%
exp_fast		2	0	50.0%	50.0%
gcd	√	2	0	50.0%	50.0%
hanoi		4	3	16.7%	66.7%

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] [4]. For procedure compression, procedures are separately compressed. Upon invocation, a procedure is uncompressed into a *procedure cache* and executed [3]. For cache-line compression, the program lies compressed in memory: a cache miss fetches the compressed instructions and decompresses them on the fly [6] [10]. The instruction width can be reduced by implementing 16-bit subsets of 32-bit ISAs [5] [9].

There is some research on reducing stack size. Variables with non-overlapping live ranges can be assigned to the same stack slot [7]. This can also be combined with code size optimization [11].

5. Future Work

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, we started to implement our algorithms as a post pass optimizer. Partial results for our basic stack size algorithm are already available in [8].

6. Conclusion

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

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. This allowed us to optimize the stack size in 70% of our benchmarks. For these 70%, we saved between 16.7% and 50% of stack space.

Often, live ranges of formal parameters or local variables can be split at recursive calls through program transformations. These transformations allowed us to optimize the stack size further. For *all* programs, including previously unoptimizable programs, we saved between 16.7% and 50% of stack space.

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] 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.
- [4] 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.
- [5] 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.
- [6] C. Lefurgy. *Efficient Execution of Compressed Programs*. PhD thesis, University of Michigan, 2000.
- [7] S. Schaeckeler and W. Shang. Stack compaction for memory constrained systems. In *IEEE International Conference on Computing and Informatics (ICoCI 2006)*, Kuala Lumpur, Malaysia, 2006.
- [8] S. Schaeckeler and W. Shang. Stack reduction of recursive programs. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2007)*, Salzburg, Austria, 2007.
- [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.