Function Inlining under Code Size Constraints for Embedded Processors

Rainer Leupers, Peter Marwedel
University of Dortmund
Dept. of Computer Science 12
44221 Dortmund, Germany
email: leupers@ls12.cs.uni-dortmund.de

Abstract– *Function inlining is a compiler optimization that generally increases performance at the expense of larger code size. However, current inlining techniques do not meet the special demands in the design of embedded systems, since they are based on simple heuristics, and they generate code of unpredictable size. This paper presents a novel approach to function inlining in C compilers for embedded processors, which aims a maximum program speedup under a global limit on code size. The core of this approach is a branch-and-bound algorithm which allows to quickly explore the large search space. In an application study we show how this algorithm can be applied to maximize the execution speed of an application under a given code size constraint.*

1 Introduction

For embedded systems based on programmable processors, C compilers play an important role in the system design process. While assembly-level programming of embedded processors has been common for quite some time, using C compilers for programming embedded processors gains more and more acceptance. Compilers permit shorter design cycles, higher productivity and dependability, and better opportunities for reuse than assembly programming. However, code generated by compilers usually implies an overhead in code size and performance as compared to hand-written assembly code. While this overhead is acceptable in general-purpose computing, the demands on compilers are different for embedded systems. In order to minimize the overhead of compiler-generated code, compilers for embedded processors have to pay higher attention to code optimization rather than high compilation speed. As a consequence, a number of code optimization techniques for embedded processors have been developed. Most of these are low-level optimizations, which exploit detailed knowledge about the processor architecture for optimizing machine code. Examples are techniques for code selection, register allocation, and scheduling [1, 2, 3], memory access optimization [4], and optimization of address computations [5, 6].

Complementary to these techniques, in this paper we present a largely machine-independent source-level code optimization, which employs *function inlining* to achieve higher performance. Function inlining is a well-known technique used in many compilers for general-purpose processors. The main idea is to replace calls to a function by copies of the function body. In this way, the function is turned into a C-level macro.

Since the overhead associated with function calls (parameter passing, call and return instructions, instruction pipeline stalls, saving and restoring register contents) is reduced, function inlining tends to increase performance. However, inlining generally also increases code size. Therefore, in order to avoid a code size explosion, only a limited set of ”small” functions within an application may be candidates for inlining.

Inlining of functions during compilation can be guided by the user via explicit ”inline” keywords in function definitions in the source code (e.g. in C++ and, as a non-standard feature, in most ANSI C compilers). Many compilers are also capable of automatic inlining, where inlined functions are selected by a set of simple, mostly local, heuristics [7, 8]. For instance, a function might be inlined, if its calling overhead appears to be larger than the time needed to execute the function body. In some compilers, e.g. in the TI ’C6x ANSI C compiler, additionally a maximum size threshold for inlined functions can be specified as a compiler option. However, all these ad hoc methods cannot guarantee that the set of selected inline functions actually lead to the maximum speedup in program execution time. Furthermore, these methods are not capable of meeting *code size constraints*, which is particularly important for embedded systems-on-a-chip with a limited amount of program memory.

The technique proposed in this paper represents a more systematic way of performing function inlining in the context of embedded system design. It is based on the assumption that when compiling an application C source code without any inlining, the resulting machine code does not completely occupy the available program memory, but that the remaining program memory space can be exploited to speed up the machine program. Given a C source code with a total of M functions, including N functions, $N \leq M$, which are candidates for inlining, our goal is to determine which subset of the N candidate functions must be inlined, such that the resulting code size does not exceed the given limit and that the performance increase is maximized. Since the number of possible solutions is 2^N, a brute-force exhaustive search is clearly infeasible, except for small values of N.

The core of our technique, therefore, is a branch-and-bound (B & B) algorithm which allows to explore the solution space in comparatively short time. This algorithm selects the subset of inlined functions while exactly minimizing the number of *dynamic* function calls (i.e., calls executed at program runtime) under a given code size constraint. Its main idea is that minimizing the number of dynamic calls approximately also minimizes the execution time. However, in general, this is not exactly true, since inlining may also increase the number of resource conflicts. Therefore, a larger code size limit is not guararanteed to result in higher performance. In order to take into account the negative effects of inlining on performance, we embed the B & B algorithm into an *interval search* over different permissible code size limits. The lower interval bound is the initial code size without any inlining, while the upper bound reflects a given *maximum* code size limit. For each limit within the interval, using a certain search granularity, the B & B algorithm is applied, and simulation is used to determine the exact performance. Finally, the best solution is emitted, which meets the maximum code size limit at the highest performance. Further details will be given in section 3.

The structure of the paper is as follows. In the next section, we describe the B & B algorithm used for minimizing the number of dynamic function calls. Section 3 shows how this algorithm, in combination with the interval search over code size limits, is used to minimize the execution time for an application. This is demonstrated for a DSP application. Our experimental results indicate that performing function inlining in a systematic way can lead to a significant performance increase at a moderate growth in code size.

2 Minimization of function calls

For a given application C source code with M functions, our technique requires the following input data.

- The subset $\{f_1, \ldots, f_N\}$, $N \leq M$, of functions which are **candidates for inlining**. Recursive functions and functions within a call cycle have to be excluded from inlining in order to avoid infinite loops. Also top-level functions, i.e., the "main" function or functions not called anywhere in the source code cannot be candidates for inlining.
- The **basic code size** $B(f_i)$ for each function f_i. These values are determined by compiling the source code once without any inlining.
- The number $D(f_i)$ of **dynamic calls** to each function f_i for a typical set of input data. This information is obtained by profiling.
- For each pair of functions (f_i, f_j) the number $C(f_i, f_j)$ of **static calls** from function f_i to function f_j, i.e., the number of occurrences of calls to f_j in the source code of f_i.
- A **code size limit** L, which is assumed to be larger than the sum of the basic size values $B(f_i)$ over all functions f_i.

We represent the set of inlined functions as a bit vector $IV = (b_1, \ldots, b_N)$, hereafter called the *inline vector*. For any function f_i, a value of $b_i = 1$ ($b_i = 0$) denotes that function f_i is (not) selected for inlining. Under the code size constraint L, we would like to minimize the number D of total dynamic functions calls in order to (approximately) minimize the execution time. Since inlined functions are never called, this value is, for a given inline vector IV, defined by

$$D(IV) = \sum_{f_i : b_i = 0} D(f_i)$$

Trivially, the theoretical minimum value of D is obtained, if all candidate functions are inlined, i.e., $IV = (1, \ldots, 1)$. However, such a solution is unlikely to meet the code size limit in practical cases. Therefore, among all 2^N possible inline vectors, we have to identify that one which minimizes D such that the total code size does not exceed L.

In order to compute the total code size for a given inline vector, it is important to consider the mutual dependence of function code size values. Consider a function f_i with $c = C(f_i, f_j)$ static calls to function f_j. If f_j is inlined, then the resulting size of f_i, initially given by its basic size $B(f_i)$, grows by c times the size of f_j. In turn, the size of f_j, which might call another function f_k that again may or may not be inlined, obviously depends on IV. Therefore, the decision to inline a certain function may have a global effect on the overall code size. This will be exemplified in the application study in section 3.

```
algorithm MIN_IV
input: inline vector $IV = (b_1, \ldots, b_N) \in \{0, 1, x\}^N$;
output: inline vector $IV' \in \{0, 1\}^N$;
begin
  if $IV$ contains no "$x$" bits then
    if $S(IV) \leq L$ and $D(IV) < D_{\min}$ then
      $D_{\min} := D(IV)$;
    end if
    return $IV$;
  end if
  $i$ := first index in $IV$ for which $b_i = x$;
  if $S((b_1, \ldots, b_{i-1}, 1, 0, \ldots, 0)) > L$ then
    return MIN_IV($(b_1, \ldots, b_{i-1}, 0, x, \ldots, x)$);
  else if $D((b_1, \ldots, b_{i-1}, 0, 1, \ldots, 1)) > D_{\min}$ then
    return MIN_IV($(b_1, \ldots, b_{i-1}, 1, x, \ldots, x)$);
  else
    $IV_0$ := MIN_IV($(b_1, \ldots, b_{i-1}, 0, x, \ldots, x)$);
    $D_0 := D(IV_0)$;
    $IV_1$ := MIN_IV($(b_1, \ldots, b_{i-1}, 1, x, \ldots, x)$);
    $D_1 := D(IV_1)$;
    if $D_1 < D_0$ then
      return $IV_1$;
    else
      return $IV_0$;
    end if
  end if
end algorithm
```

Figure 1: *Branch-and-bound algorithm*

For a given IV, the total code size $S(IV)$ can be computed as follows:

$$S(IV) = \sum_{i=1}^{M} S_i(IV)$$

$$S_i(IV) = B(f_i) + \sum_{f_j : b_j = 1} C(f_i, f_j) \cdot S_j(IV)$$

The recursion in the definition of $S_i(IV)$ terminates at the *leaf functions* which do not contain any function calls. Note that the code size computed in this way is not exact but represents an estimation, since the detailed effects of function inlining on code size are only known after code generation.

Given the definitions of $D(IV)$ and $S(IV)$, we use a branch-and-bound search to efficiently compute the optimum inline vector meeting the constraint $S(IV) \leq L$ at a minimum D value. We start with an initial vector $IV = (x, \ldots, x)$, where $b_i = x$ denotes that the inlining of function f_i is not yet decided. Then, for $i = 1, \ldots, N$, all bits b_i in IV are determined one after another. The B & B algorithm, whose pseudo code is shown in fig. 1, is based on the following problem analysis.

Let $IV = (b_1, \ldots, b_{i-1}, x, \ldots, x)$ be the current inline vector, where the values of $b_1, \ldots, b_{i-1}$ have already been fixed. Next, we would like to determine b_i, i.e., the leftmost "x" bit in IV. Let variable $D_{\min}$ denote the minimum number of dynamic function calls found so far, such that the corresponding solution meets the code size constraint $S(IV) \leq L$. Initially, $D_{\min}$ is set to

$$D_{\min} = \sum_{i=1}^{M} D(f_i)$$

which is equal to the number of dynamic function calls without any inlining. Now, four different cases may occur:

1. If there is no further "x" bit in IV, then we compute the total code size $S(IV)$ and the number of function calls $D(IV)$ for the solution represented by IV. If $S(IV) \leq L$ and $D(IV) < D_{\min}$, then a new valid minimum has been found, and we set $D_{\min} := D(IV)$.
2. Otherwise, if $b_i = x$, we check whether function f_i can be inlined without violating the code size limit by temporarily setting $b_i = 1$. A lower bound on total code size for $b_i = 1$ is given if, for $j = i+1, \ldots, N$, all remaining "x" values in IV are set to zero, i.e., no further function is inlined. The corresponding inline vector is $IV_1 = (b_1, \ldots, b_{i-1}, 1, 0, \ldots, 0)$. If $S(IV_1) > L$, then inlining f_i cannot result in a valid solution, independent of $(b_{i+1}, \ldots, b_N)$. Thus, the corresponding part of the search space can be cut off without loss of optimality by setting $b_i = 0$. Note that the test $S(IV_1) > L$ also ensures that *any* invocation of algorithm MIN_IV can only return a valid solution.
3. If $b_i = 1$ could not be excluded, then we check whether a new minimum D value would be possible if f_i were *not* inlined. This can be determined by temporarily setting $b_i = 0$ and, for $j = i+1, \ldots, N$, setting all remaining "x" values in IV to one. The corresponding inline vector is $IV_0 = (b_1, \ldots, b_{i-1}, 0, 1, \ldots, 1)$. Then, the value $D(IV_0)$ provides a lower bound on the number of dynamic calls. If $D(IV_0) > D_{\min}$, then not inlining f_i cannot improve the best solution found so far, and the case $b_i = 0$ can be cut off from the search space. In this case, b_i is set to one and the search continues for the next undecided bit b_{i+1} in IV.
4. In the worst case, pruning the search space by computing lower bounds is not possible, so that both alternatives $b_i = 0$ and $b_i = 1$ have to be evaluated in detail. First, we try $b_i = 0$ and recursively compute the corresponding minimum number of dynamic function calls D_0. Let IV_0 be the resulting inline vector. Next, b_i is set to 1, and the corresponding minimum number of function calls D_1 is recursively computed together with the resulting inline vector IV_1. If $D_1 < D_0$ then $b_i = 1$ is the best solution and is also guaranteed to be valid w.r.t. the code size limit L, due to the test performed in step 2. Otherwise, we need to set $b_i = 0$. Depending on the setting of b_i, either IV_0 or IV_1 are returned as the optimal inline vector.

The worst case complexity of this branch-and-bound algorithm is exponential in N. However, in many cases pruning the search space in steps 2 and 3 is possible. This permits to optimize function inlining also for relatively large values of N. This will be demonstrated in the next section.

3 Minimization of execution time

In order to evaluate the proposed technique, we have performed an application study for a complex DSP application from the area of mobile telephony: a GSM speech and channel encoder. This application is specified by 7,140 lines of C code, comprising 126 different functions. Out of these, 26 functions are dedicated "basic" functions for certain arithmetic operations. These basic functions are relatively small and are frequently called from other functions. Therefore, the basic functions were the natural candidates for function inlining. As a target processor, we have used the TI 'C6x, a VLIW DSP with 8 functional units, together with TI's corresponding ANSI C compiler and instruction set simulator.

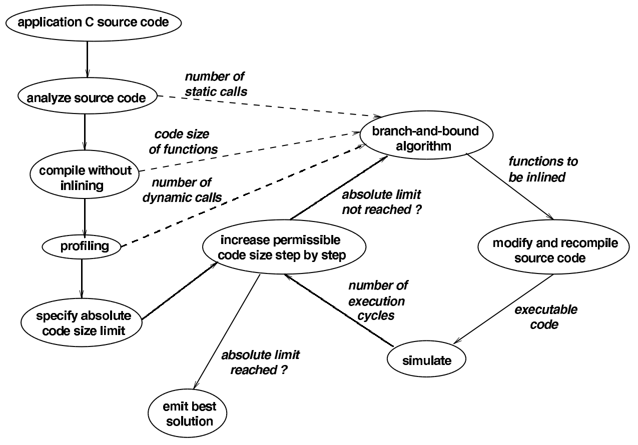
application C source code
analyze source code
number of static calls
code size of functions
compile without inlining
number of dynamic calls
profiling
specify absolute code size limit
branch-and-bound algorithm
absolute limit not reached ?
increase permissible code size step by step
functions to be inlined
modify and recompile source code
number of execution cycles
executable code
simulate
absolute limit reached ?
emit best solution

Figure 2: *Methodology overview*

An overview of our methodology is shown in fig. 2. The first step was to use a simple source code analysis tool in order to determine the number of static calls $S(f_i, f_j)$ from f_i to f_j for each pair of functions. Next, we compiled the source code without function inlining in order to determine the basic code size $B(f_i)$ for all functions f_i. Finally, we performed profiling on a given set of input speech data in order to obtain the number of dynamic calls $D(f_i)$ for all functions f_i. Without inlining, we obtained a total code size of 67,820 bytes. The initial number of execution cycles as determined by simulation was 27,400,547.

After that we analyzed, whether inlining of some of the 26 basic functions under a code size limit would result in higher performance. We have arbitrarily allowed for a maximum code size increase of 50 % as compared to the initial size, and we used the B & B algorithm to explore the vast search space of 2^{26} possible solutions. As mentioned earlier, the B & B algorithm minimizes the number of dynamic function calls, but this does not necessarily also minimize the number of execution cycles. This means that, in principle, all code size limits within the interval 100 % (the initial code size) to 150 % have to be considered to find the actual optimum w.r.t. performance. Since this obviously cannot be accomplished within reasonable time, we have performed an evaluation for the interval between 100 % and 150 % in steps of 5 %. For each code size limit, we ran the B & B algorithm in order to determine the optimum set of inlined functions. Then, the C source code was modified accordingly by tagging the selected functions with an "inline" keyword, thereby enforcing inlining of those functions by the compiler, and the source code was re-compiled and simulated.

Our results are summarized in table 1. The first column gives the code size limit in percent relative to the initial solution without inlining. The second and third columns show the absolute and relative number of dynamic function calls, respectively, which monotonically decrease with the increasing code size limit. Columns 4 and 5 show the code size as estimated by function $S(IV)$ and the real code size, while column 6 gives the estimation error. As can be seen, the estimation is highly accurate, with a maximum error of 3 %. More important, in no case did the real code size exceed the estimated size.

Columns 7 and 8 account for the absolute and relative num-

size limit (%)	calls (absolute)	calls (%)	est. size	real size	error (%)	cycles (absolute)	cycles (%)	CPU
100	10,292,056	100	–	67,820	–	27,400,547	100	–
105	7,618,479	74	71,200	70,284	1	24,095,022	88	62
110	5,893,530	57	74,536	72,876	2	19,560,628	71	117
115	4,984,329	48	77,976	77,796	1	20,190,858	74	186
120	4,403,360	43	81,372	80,772	1	20,518,980	75	348
125	3,892,613	38	84,768	82,636	3	18,235,114	67	351
130	3,427,558	33	88,148	87,908	1	18,527,926	68	521
135	2,414,683	23	91,544	89,796	2	18,416,065	67	696
140	2,385,409	23	93,812	91,940	2	18,750,981	68	933
145	1,872,297	18	98,320	97,956	1	18,796,095	69	1138
150	1,797,790	17	101,716	100,484	1	19,136,175	70	1202

Table 1: *Experimental results for GSM encoder application on a TI 'C6x*

size limit (%)	inline vector (functions 1-26)
100	00000000000000000000000000
105	00100000001100001110111111
110	10111001011100001111111111
115	10110000000001001000111001
120	10110100101000100110111101
125	10110000001010000100111101
130	00110000000010100100111000
135	10110010001110101110111101
140	10111011111110101111111111
145	10110110101010100110111101
150	10110110000010110110111101

Table 2: *Inline vectors computed by B & B algorithm*

ber of instruction cycles as determined by simulation. Although the number of function calls decrease with an increased code size limit, this does not exactly also hold for the number of execution cycles. For a limit of 150 %, the execution time is reduced to 70 % of the original value, but the absolute minimum (67 %, marked line in table 1) is achieved for a limit of only 125 %. Beyond this value, the negative effects of function inlining due to a larger amount of resource conflicts become predominant. As a consequence, a "saturation" takes place, where an increase in code size does not lead to higher performance. Still, the execution time beyond a code size of 125 % does not deviate much from the optimum, so that the B & B algorithm could also be used as a stand-alone optimization to obtain a close-to-optimum solution without the need for repeated simulation.

Finally, column 9 gives the CPU seconds needed for executing the B & B algorithm on a SUN Ultra-1 workstation. The runtime grows with the code size limit, since a less tight limit removes opportunities for pruning the search space at an early point of time. For the maximum limit of 150 %, the required CPU time was about 20 minutes. This appears to be high, but the actual bottleneck in our application study was the time required by the TI 'C6x simulator, which ranged between one and two hours in each case.

Besides these results, it is also interesting to take a look at the inline vectors computed by the B & B algorithm as shown in table 2. The detailed bit values are less important, but it can be observed that the inline vectors tend to change at many bit positions from step to step. This means that among the set of candidate functions there are hardly functions for which inlining pays off independent of the code size, but that the optimum set of inlined functions are globally influenced by the concrete code size limit. This observation motivates the use of the proposed B & B algorithm instead of a possible faster, but less effective, local optimization approach.

4 Conclusions

C compilers should be used for software development for embedded processors in order to replace assembly language programming. However, compilers will only gain acceptance if they take into account the special demands in the design of embedded systems, such as very high code quality and limited code size. This requires special code optimization techniques for embedded processors. In this paper we have described a new technique for optimized function inlining which, in contrast to techniques used in compilers for general-purpose processors, is capable of maximizing performance while meeting a global code size constraint. In our application study, the net effect was a performance increase of 33 % at an increase in code size of 25 %. Naturally, the detailed results depend on the application and the target processor. However, our results indicate that high-level code optimizations like function inlining should definitely be considered equally important to complementary optimization techniques working at the assembly code level.

References

[1] C. Liem, T. May, P. Paulin: *Instruction-Set Matching and Selection for DSP and ASIP Code Generation*, European Design and Test Conference (ED & TC), 1994, pp. 31-37

[2] G. Araujo, S. Malik: *Optimal Code Generation for Embedded Memory Non-Homogeneous Register Architectures*, 8th Int. Symp. on System Synthesis (ISSS), 1995, pp. 36-41

[3] S. Liao, S. Devadas, K. Keutzer, S. Tjiang: *Instruction Selection Using Binate Covering for Code Size Optimization*, Int. Conf. on Computer-Aided Design (ICCAD), 1995, pp. 393-399

[4] A. Sudarsanam, S. Malik: *Memory Bank and Register Allocation in Software Synthesis for ASIPs*, Int. Conf. on Computer-Aided Design (ICCAD), 1995, pp. 388-392

[5] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang: *Storage Assignment to Decrease Code Size*, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1995

[6] R. Leupers, P. Marwedel: *Algorithms for Address Assignment in DSP Code Generation*, Int. Conf. on Computer-Aided Design (ICCAD), 1996

[7] S.S. Muchnik: *Advanced Compiler Design & Implementation*, Morgan Kaufmann Publishers, 1997

[8] R. Morgan: *Building an Optimizing Compiler*, Butterworth-Heinemann, 1998