

Function Inlining with Code Size Limitation in Embedded Systems

Xinrong Zhou, Johan Lilius, and Lu Yan

Turku Centre for Computer Science, Abo Akademi University, Finland

Abstract: *Function inlining is a widely known technique that has been adopted in compiler optimization research domain. Inlining functions can eliminate the overhead which is resulted from function calls, but with inlining, the code size also grows unpredictably; this is not suitable for embedded processors whose memory size is relatively small. In this paper, we introduce a novel function inlining approach using a heuristic `rebate_ratio`; functions to be inlined are selected according to their `rebate_ratios` in a descending way. This kind of code optimization operation works at the source code level. Compared with other algorithms, ours are easier to implement. Our target is to get an optimal result of function inlining which attempts to achieve the maximum performance improvement while keeping the code size within a defined limit.*

Keywords: *Function inlining, embedded processors, graph, complexity.*

Received April 2, 2004; accepted July 1, 2004

1. Introduction

Nowadays, more and more people prefer to use C compiler rather than assembly one for programming embedded processors. In a C program, the most frequently accessed parts are often put together into functions. It makes the programs more dependent and more readable, but an excessive use of functions may degrade program performance. When calling a function, the system should save all the values of current registers, pass the parameters and allocate stack for local variables. In processors which support pipeline, the actual function call and return may result in a significant number of instruction pipeline stalls. Function inlining replaces a function call with the body of function; it has the effect of removing all the overheads mentioned above. [1, 2, 3, 4, 6] Obviously, the performance of the system can be improved in some ways, but inlining functions does not come for free. One of its negative effects is the unpredictable code size; which is intolerable for embedded processors whose memory space is limited. During the past years, lots of code optimization techniques have been developed. Many of them are low level optimizations, which are dependent on processor architecture. For example, code selection, register allocation [5, 10], and memory access optimization [12]. These works focus on how to get a performance enhancement, less attention was put on the code size. Leupers brought out a machine-independent source-level code optimization algorithm, named OptinlineVector, which aims at embedded processors and employs function inlining to achieve higher performance [5, 7]. In OptinlineVector algorithm the element b_i in the inline vector IV is used to indicate

whether function f_i is inlined or not, all the functions in program are checked. OptinlineVector algorithm can find the optimal solution of function inlining, but the time and memory space it needs are huge. The worst case complexity of OptinlineVector algorithm is exponential to N , where N is the total number of functions in a program. In this paper, we present a new approach to function inlining which works at the source code level as well. The time and memory space needed in worst case is the cube of N .

The remainder of this paper is organized as follows. Section 2 illustrates the system model of function inlining. Our new algorithm is explained in detail in section 3. Section 4 makes a brief analysis of our algorithm. The last section concludes this paper and points out our future work.

2. System Model for Function Inlining

In normal systems, performance enhancement is the main target, the negative effect of code expansion which is brought by function inlining does not attract more attention, but in embedded processors, code expansion becomes a serious problem. An oversized code is intolerable. In order to control the code bloating problem of inlining, we should inline selectively. Leupers et al used branch-and-bound algorithm to determine which function to inline [7]. Although their result is an optimal one, the time and space their algorithm needs are huge. In our method, we use heuristic to do the same job. The benefit using heuristic depends on the execution frequency of the inlined function. The more it is called, the better improvement it will achieve. We introduce a concept, named `rebate_ratio`; it is used as an inlining heuristic

variable. Inlining a function with a high `rebate_ratio` will get a better performance than inlining a low `rebate_ratio` function.

The definition of `rebate_ratio` is:

$$rebate_ratio = \frac{function_calling_frequency}{code_size_increased} \quad (1)$$

The function calling frequency is direct proportion to performance improvement while code size expansion is the other way round. Note that, in some case, `code_size_increased` may equal to zero, which means when inlining that function, the code size does not change. We assign a maximum value to the `rebate_ratio` of this function and inline them before inlining other functions.

The system model of function inlining is described as follows. For a given C program, we use a graph $G = (V, E)$ to represent the function call structure inside it. Each node in V represents one function f_i and each edge $e = (v_i, v_j) \in E$ means function f_i calls function f_j . Each node v_i has a two-tuple attributes $v_i: (B_i, R_i)$, B_i denotes the real size of function f_i , R_i is the `rebate_ratio` of function f_i . Attribute R_i is used as a priority indicator of our queue operating, the smaller R_i is the higher possibility it will be at the head of a queue, which means the higher possibility to be inlined. Each edge e_i has a weight w_i which denotes the times function f_i calls f_j .

The total sum of all the nodes' weight in V is the estimation of total code size of the given C program. It can be seen, the code size calculated in this way is not precise since the detailed code size is only known after code generation. Algorithms using similar method to calculate code size have already shown that this estimation appears to be sufficiently accurate in practice [5, 6, 7]. The function inlining problem is now translated to a graph operation problem, what we will do is to present a method to realize the following work:

- Input: $G = (V, E)$ and a global code size limit L .
- Output: $G' = (V', E')$ which $|V'|$ reaches its minimum value while:

$$\sum_{i=1}^{|V'|} B(v_i) \leq L$$

where $|V'|$ is the number of nodes in Graph G' .

3. Minimizing Function Calls

As the number of function calls in a program decreases, the performance increases. Once a function is inlined, the corresponding operation in graph is that the node representing that function is deleted. When the code size of the inlining function increases, the change in graph is that the weight of deleted node's parent node also increases. Since the code size has an upper bound and we wish to inline as many function

calls as possible, the operation to the graph is trying to delete as many nodes as possible while keeping the total sum of all the remained nodes' weight not larger than the limit value.

We inline the function calls in a `rebate_ratio` decreasing way, in another word, we inline first the function whose `rebate_ratio` is the largest and then inline the second largest and so on. When no more function call can be inlined with the total code size smaller than the upper bound, the work is done.

Before we start inlining functions, there is some preparation work to be done. First, we must use a source code tool to find out the number of calls $w(e(v_i, v_j))$ from function f_i to f_j . Next, to compile the source code without function inlining to determine the code size $B(v_j)$ of each function.

Usually there are lots of loops in a program, if an inlined function is in a loop, the amount of code size increased is equal to the code size of this inlined function, not n times of code size (suppose n is the number of loop repetition), i. e., `rebate_ratio` is $n/code_size$ not $n/(n * code_size)$.

The benefit of inlining a function in a loop is the same as inlining n times of a function whose code size is n times smaller. So in our algorithm, we assign an equal priority to the two functions, which means their `rebate_ratios` are the same.

If a function calls another one f_k both in loops and outside loops, we derive a new node v_n , where $B(v_n) = B(v_k)$, $R(v_n) = R(v_k) / n$, here n is the iteration of the loops, the new node is connect with all of node v_k 's parent and child nodes. The weights of the new derived edges are defined as follows.

$$\begin{aligned} \forall v_p \in parent(v_k) \quad \forall v_c \in child(v_k) \\ w(e(v_p, v_n)) = 1 \quad w(e(v_n, v_c)) = w(e(v_k, v_c)) \end{aligned}$$

where, the function that the new node represents is a copy of the function in loops. The weight $w(e(v_p, v_k))$ is also reduced to the number of calls outside loop. Thus, we change the nodes for functions in loops into normal ones. Figure 1 is an example of node deriving for functions in loop. Function f_1 calls f_3 w_3 times, $w'_3 = w_3 - n$ times are not in loops, the algorithm derives a new node v'_3 , derives also an edge $e(v_1, v'_3)$, the edge's weight is 1.

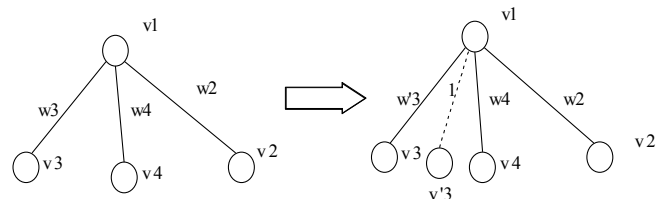


Figure1. Node deriving for functions in loops.

There are two different situations when function f_i inlines function f_j , first, v_j is a leaf in the graph shown in Figure 2, we delete both the node v_j and the edge

$e(v_i, v_j)$, the weight of node v_i changes to a new value $B'(v_i) = B(v_i) + W(e(v_i, v_j)) * B(v_j)$, if more than two functions call f_j , the weight values of all these functions should also be modified and the edges be deleted.

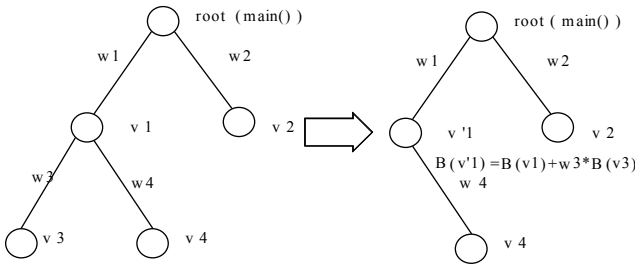


Figure 2. Deleting a leaf node.

Second, function f_j is not a leaf in the graph, node v_j is deleted, all its parent nodes' weights are also updated, the edges connected with v_j are deleted, v_j 's parent nodes are connected with v_j 's children nodes. The newly appeared edges also have new weights. For example, when root function inlines f_1 , as shown in Figure 3, node v_1 is deleted, so do all the edges connecting with v_1 , node v_3, v_4 's parents are changed to the root, two new edges are derived, the weight values are $w_1 * w_3$ and $w_1 * w_4$ respectively.

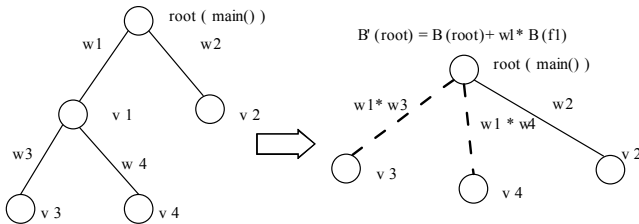


Figure 3. Deleting a non-leaf node.

Our method to inline functions consists of two steps. First, we process the functions in loops using algorithm Loop_Node_Process, shown in Figure 4. We search the whole program and find out the functions in loops, and then change the corresponding nodes to new ones which are the same as the nodes representing outside loop functions, when the changing work finishes, we set up a queue and sort the queue.

Second, we inline the functions according to algorithm Mini_Func_Call, shown in Figure 5. Since the queue has already been sorted in a rebate_ratio descending way, the most suitable function to be inlined is the one which is represented by the node in the head of the queue. We inline the function and delete the node from the queue. After inlining, the rebate_ratio values of its parent nodes are also changed, so we sort the queue again and ensure the first node in the queue has the largest rebate_ratio. When all the nodes in the queue are deleted, the algorithm is terminated.

Loop_Node_Process (V, E)

1. $V' \leftarrow Find_Nodes_in_Loop(V)$
2. for (v_i in V') do
3. create a new node v_k
4. $B(v_k) \leftarrow B(v_i)$
5. $R(v_k) \leftarrow R(v_i)/n$ {increase the rebate_ratio of the new node}
6. insert_queue(Q, v_i) {insert the new node into queue Q}
7. for (v_j in child(v_i)) do
8. $E \leftarrow E + e(v_k, v_j)$ {add edge $e(v_k, v_j)$ to E}
9. $w(e(v_k, v_j)) \leftarrow w(e(v_i, v_j))$
10. end for
11. for (v_j in parent(v_i)) do
12. $E \leftarrow E + e(v_j, v_k)$
13. $w(e(v_j, v_k)) \leftarrow 1$
14. $w(e(v_j, v_i)) \leftarrow w(e(v_j, v_i)) - n$ {update the number v_j calls v_i }
15. update $R(v_i)$
16. if ($w(e(v_j, v_i)) = 0$) then
17. $V \leftarrow V - v_i$ {delete node v_i from the graph}
18. delete_queue(Q, v_i) {get rid of the node from queue Q}
19. for (v_c in child(v_i)) do
20. $E \leftarrow E - e(v_i, v_c)$ {delete edges connected to the child nodes}
21. end for
22. end if
23. end for
24. end for
25. sort_queue(Q)
26. return ($V + V', E$)

Figure 4. Algorithm loop_node_process.

4. Algorithm Analysis

Although heuristics are also used to find the optimal result of function inlining, unlike other ones, the value of the heuristic – rebate_ratio in our algorithms is not fixed, it keeps on changing whenever a function's code size varies.

If we inline a function, the benefit we get is that we eliminate the overhead which is brought by setting up the call stack, passing parameters etc, the side-effect is the expansion of code size. Heuristic – rebate_ratio is an indicator of the combination of the benefit and the side-effect. As shown in the algorithm Mini_Func_Call, we select the one whose rebate_ratio value is the largest when we inline a function, in this way, we can get more performance improvement than inlining other functions. If a function has inlined other functions, its code size may increases, the side-effect of being inlined enlarges, its rebate_ratio decreases to a

smaller value, which means it gives the priority of selection to other functions.

Inside our algorithm, n represents the number of functions. There are 3 level iteration, line 2, 11, 19 in algorithm `Loop_Node_Process` and line 3, 12, 14 in algorithm `Mini_Func_Call`, the worst case of the time and space complexity of our algorithm is $O(n^3)$, if there are no circles in the graph, i. e., the graph is a family tree, the numbers of parent and child nodes are $O(\log n)$, the time we need reduces to $O(n \log^2 n)$. In the most ideal situation, when the graph degenerates to a line, the complexity is equal to $\Theta(n)$.

The exception of our algorithm is described as following:

If there exists two adjacent nodes v_i and v_j in queue Q , function f_i has a larger `rebate_ratio`, when inlining function f_i , the code size is over the limit, i. e.,

$$\sum_{k=1}^{|V-v_i|} B(v_k) + \sum_{v_p \in \text{parent}(v_i)} (w(e(v_p, v_i)) \times B(v_i)) > L \quad (2)$$

So, we give up function f_i and select function f_j , and we go on running until it reaches the final, but the total performance enhancement gained from inlining function f_j to end is not as good as inlining function f_i in part of its parent nodes' calls, this phenomena may occur in nest. One solution to this problem is that we derive as many sibling nodes as possible when the above situation is detected, the newly born nodes have the same `rebate_ratio`, and they join the queue Q waiting for selection.

5. Conclusion and Future work

The small memory space of embedded processors requires applications keep a sophisticated tradeoff between the program code size and system performance. Nowadays' heuristics inlining techniques do not meet such a demand. In this paper we present a code optimization technique which works at the source code level. It can minimize the number of function calls by inlining proper subset of functions under a code size constraint.

Like other algorithms, we need profiling to get the exact number of functions to inline. The repetition times in recursive loops, repeat/until and while statements are uncertain, when processing these loops, we give a rough estimation. Sometimes inlining functions in these loops can give significant savings; one of our future works is to handle this situation precisely. Some functions which are small in size but have many local variables may have a negative effect on the execution time when inlined, more work will also need to focus on solving these problems in the near future.

Mini_Func_Call (V, E, L)

1. $G(V, E) \leftarrow \text{Loop_Node_Process}(V, E)$
2. $Q \leftarrow \text{create a queue, } \{ \text{queue } Q(v_1, v_2, \dots, v_n), v_i \in V, |V| = n \}$
 $\text{sort_queue}(Q), \{ \forall v_i, v_j \in Q, (i < j), R(v_i) > R(v_j) \}$
3. *while* (Q is not empty) *do*
4. $v_i \leftarrow \text{first element in } Q$
5. *delete* v_i *from* Q
6. *if*
 $(\sum_{k=1}^{|V-v_i|} B(v_k) + \sum_{v_p \in \text{parent}(v_i)} (w(e(v_p, v_i)) \times B(v_i)) \leq L)$
then
 $\{ \text{if code size is within constraint after inlining function } f_i \}$
7. $V \leftarrow V - v_i$ $\{ \text{delete node } v_i \text{ from the graph} \}$
8. *for* (v_c in child (v_i)) *do*
9. $E \leftarrow E - e(v_i, v_c)$ $\{ \text{delete edges connected to the child nodes} \}$
10. *end for*
11. *for* (v_p in parent (v_i)) *do*
12. $B(v_p) \leftarrow B(v_p) + w(e(v_p, v_i)) * B(v_i);$
 $\{ \text{modify parent node's code size} \}$
13. *update* $R(v_p)$
14. *for* (v_c in child (v_c)) *do*
15. $E \leftarrow E + e(v_p, v_c)$ $\{ \text{derive new edges} \}$
16. $w(e(v_p, v_c)) \leftarrow w(e(v_p, v_i)) * w(e(v_i, v_c))$
17. *update* $R(v_c)$
18. *end for*
19. *end for*
20. *sort_queue* (Q)
21. *end if*
22. *end while*
23. *return* (V, E)

Figure 5. Algorithm `mini_func_call`.

References

- [1] Araujo G. and Malik S., "Optimal Code Generation for Embedded Memory Non-Homogeneous Register Architecture," in *Proceedings of the 8th International Symposium on Synthesis*, pp. 36-41, 1995
- [2] Ayers A., Gottlieb R., and Schooler, "Aggressive Inlining," in *Proceeding of ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1997.
- [3] Davidson J. W. and Holler A. M., "A Study of a C Function Inliner," *Software Practice Experience*, vol. 18, no. 8, pp. 775-790, 1988.
- [4] Davidson J. W. and Holler A. M., "Subprogram Inlining: A Study of its Effects on Program

- Execution Time,” *IEEE Transactions on Software Engineering*, vol. 18, no.2, pp. 89-102, 1992.
- [5] Leupers R., *Code Optimization Techniques for Embedded Processors*, Kluwer Academic Publishers, 2000.
- [6] Leupers R. and Marwedel P., “Algorithms for Address Assignment in DSP Code Generation,” in *Proceedings of IEEE/ACM International Conference on Computer Aided Design*, 1996.
- [7] Leupers R. and Marwedel P., “Function Inlining under Code Size Constraints for Embedded Processors,” in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, 1999.
- [8] Liao S., Devadas S., Keutzer K., and Tjiang S., “Instruction Selection Using Binate Covering for Code Size Optimization,” in *Proceedings of the International Conference on Computer-Aided Design*, pp. 393-399, 1995.
- [9] Liao S., Devadas S., Keutzer K., Tjiang S., and Wang A., “Storage Assignment to Decrease Code Size,” in *Proceedings of Programming Language Design and Implementation*, 1995.
- [10] Liem C., May T., and Paulin P., “Instruction-Set Matching and Selection for DSP and ASIP Code Generation” in *Proceedings of the European Design and Test Conference*, pp. 31-37, 1994.
- [11] Muchnik S. S., *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, 1997.
- [12] Sudarsanam A. and Malik S., “Memory Bank and Register Allocation in Software Synthesis for ASIPs,” in *Proceedings of the International Conference on Computer-Aided Design*, pp. 388-392, 1995.



Xinrong Zhou is an assistant professor at the Department of Computer Science, Abo Akademi University, Finland and an associate professor at the Computer Engineering Department, Huazhong University of Science and Technology, China. He received his PhD degree in computer engineering from Huazhong University of Science and Technology, 1997. He was also a visiting professor at the University of Nebraska-Lincoln in 2003. Dr. Zhou is a member of IEEE, TFCC and CFTS. He has more than 8 years of experience in computer architecture research area, especially on storage system. He is an associate director of National Storage Lab, China. His current research interests are embedded systems and grid computing.



Johan Lilius is the director of Computer Science Department, Abo Akademi University, Finland. He leads the Embedded Systems Lab in Turku Centre for Computer Science (TUCS). His research interests cover a variety of topics. In general, he leads the research in state-based modeling techniques and their use in the design of embedded systems in the department. Currently, his research directions focus on UML and its use in the design of correct embedded systems.



Lu Yan is a research fellow at the Distributed Systems Design Lab, Turku Centre for Computer Science (TUCS) and a PhD fellow at the Department of Computer Science, Abo Akademi University, Finland. He received his BSc in computer science from Beijing University, China in 2000 and his MSc in computer science from Abo Akademi University, Finland in 2002. He was a visiting professor of ESIGELEC (École Supérieure d'Ingénieurs généralistes) and ESC Rouen (École Supérieure de Commerce de Rouen), France in 2004. He is a member of IEEE, BCS and FME. His current research interests include pervasive and global computing.