IT 09 004
Examensarbete 30 hp
Februari 2009

Evaluation of Inlining Heuristics in Industrial Strength Compilers for Embedded Systems

Pär Andersson

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Evaluation of Inlining Heuristics in Industrial Strength Compilers for Embedded Systems

Pär Andersson

Function inlining is a well known compiler optimization where a function call is substituted with the body of the called function. Since function inlining in general increases the code size one might think that function inlining is a bad idea for embedded systems where a small code size is important. In this thesis we will show that function inlining is a necessary optimization for any industrial strength C/C++ compiler . We show that both the generated code size and execution time of the application can benefit from function inlining. We study and compare the inlining heuristics of 3 different industrial strength compilers for the ARM processor architecture.

We also present a new inlining heuristic which makes inlining decisions based on several different properties. The study shows that the inlining heuristic proposed in this thesis is able to generate both smaller and faster code compared to the other inlining heuristics studied.

Handledare: Jan-Erik Dahlin
Ämnesgranskare: Sven-Olof Nyström
Examinator: Anders Jansson
IT 09 004
Sponsor: IAR Systems AB

Tryckt av: Reprocentralen ITC

Contents

1 Introduction **1**
- 1.1 Inlining heuristics . . . 2
- 1.2 Terminology . . . 3
  - 1.2.1 Module . . . 3
  - 1.2.2 Call Graph . . . 3
  - 1.2.3 Control Flow Graph . . . 3
  - 1.2.4 Dominators . . . 3
  - 1.2.5 Unreachable-Code Elimination . . . 3

2 Inlining heuristics in studied compilers **5**
- 2.1 IAR Inliner . . . 5
- 2.2 RealView Inliner . . . 6
- 2.3 GNU Compiler Collection (GCC) Inliner . . . 6

3 A new inliner **7**
- 3.1 Ideas for a new inliner . . . 7
  - 3.1.1 Selective inlining . . . 7
  - 3.1.2 Heuristics for call sites . . . 8
  - 3.1.3 Unreachable-Code Elimination . . . 8
  - 3.1.4 Termination of inlining . . . 9

4 Implementing the new inliner **10**
- 4.1 Analyzing functions for parameter dependency . . . 10
- 4.2 Building the edge queue . . . 11
- 4.3 Inline driver . . . 12
  - 4.3.1 Updating data structures after inlining of one call site . . 13
  - 4.3.2 User interaction . . . 15
- 4.4 Avoid getting stuck in local parts of the call graph . . . 15

5 Test suite **16**
- 5.1 Applications . . . 16
  - 5.1.1 Dinkum . . . 16
  - 5.1.2 Open Physics Abstraction Layer (OPAL) . . . 17
  - 5.1.3 Persistence of Visions Ray-tracer (POV-Ray) . . . 17
  - 5.1.4 Turing Machine Simulator . . . 17
  - 5.1.5 SPEC2000 CPU . . . 17
- 5.2 What we measure . . . 17

6 Test framework **19**
- 6.1 Tools and target . . . 19
  - 6.1.1 Simulator . . . 19
  - 6.1.2 Hardware . . . 19

7 Measurements **20**
- 7.1 Intermediate code vs generated code . . . 20
- 7.2 Compilers . . . 20
- 7.3 Results . . . 21

8 Conclusions and future work **23**

A Appendix **24**
A.1 IAR New Inliner Code growth 25
A.2 IAR New Inliner Cycles used 27
A.3 All inliners together 29

References **31**

1 Introduction

Function inlining (Procedure integration, Inline substitution) is an interprocedural optimization where we replace a function call by the body of the called function. This is an optimization that is used to increase the execution speed of the program, you would trade larger size of the program for less execution cycles.

Some of the reasons why we should expect a speed increase are because of the fact that when we inline a function call we remove the overhead of calling a function. Another reason is that when we integrate the body of the called function into the callee we will expose more code for other global optimizations to take advantage of. The reasons why we should expect a larger size is because we are likely to get multiple copies of the same function.

So if we want a faster program we could just inline all calls and therefore get it to run as fast as possible? No, there are many reasons why we should be careful when choosing to inline. Too much inlining could for example make the body of the caller grow to an unmanageable size which would result in unreasonable long compilation time. There are also effects of too much inlining that can have a negative impact on the execution speed of the program [15]. Too much inlining could for example result in more resource conflicts which makes it harder for the register allocator to do a good job which in the end would lead to more variable spills. Another example is where caller fits in the L1 instruction cache before the inlining of any calls but grow too large after inlining so that it no longer fits in the L1 instruction cache.

The compilers we will study in this thesis are targeted for embedded systems, more specifically we will use an ARM[1][1] based back-end. Therefore we should have in consideration that size is often a crucial part when building embedded applications. For a desktop application we seldom care much about the size, if we can make the application faster by using inline substitution we probably won't care how much the application grows, it's just some more space occupied on our hard drive.

For embedded applications it's a completely different picture, if the program grows above some limit it could result in the need of using different hardware with more memory, which in the end would lead to higher production costs. When we look at the results of code growth in the thesis we should therefore have this in mind, an expansion of more than 20% in the context of this thesis is most likely too much.

The goal of this thesis is to evaluate the existing inliner in IAR Systems C and C++ compiler (ICC), compare it with competitors RealView and GNU GCC, invent a new inliner and implement it, evaluate the new inliner.

[1]ARM1136JF-S

1.1 Inlining heuristics

There are properties of a function and function calls that we probably should take into consideration when deciding if we should inline or not [16, 17].

1. Size of the function (smaller is better).
2. Number of calls to the function.
3. If the function is called inside a loop.
4. A particular call contains constant-valued parameters.
5. If there is only one call site it can be inlined (needs to be static, address never taken, and could affect register pressure/cache in a bad way).
6. Size of the function is smaller than the overhead size of the function call.
7. If a particular call site to the function represents a large fraction of all call to the function then that particular call site can be inlined whereas the other call sites will not be inlined (need static program analysis or program profiling data).
8. Stack frame size (activation record size) of the function.
9. Language specific keywords such as the *inline* keyword in C++ (and C99).
10. Recursive functions, if inlined when do we stop?

Using these properties one can create a heuristic for inlining, for example Morgan [16] suggests one heuristic based on the size of the function being inlined, the number of call sites, and the frequency information for the calls. If a function is small enough (number 6 above) it is always inlined. If the function is larger than that but has a small number of call sites it can also be inlined.

1.2 Terminology

1.2.1 Module

In C or C++ this is the input to the compiler after preprocessing an input source file. One `.c` or `.cpp` file is regarded as one module. Inlining across modules is only possible if we do inlining in the linker or if the compiler is presented with more than one source file at the time (perhaps merged together to one big module). In ICC, the latter is called Multi-File Compilation (MFC).

1.2.2 Call Graph

The Call Graph (CG) of a module is a directed multi-graph `G=(V,E)` where each node in `V` is a function. An edge in `E` occurs between two nodes `F1` and `F2` if there is a call from `F1` to `F2`. Since it is a multi-graph it is possible that several edges between `F1` and `F2` occur. The CG could also include nodes which are functions that are externally declared, these nodes are however not candidates for inlining.

1.2.3 Control Flow Graph

The Control Flow Graph (CFG) for a function is represented as a directed graph `G=(V,E)` where each node in `V` is a basic block. An edge in `E` occurs between two nodes `B1` and `B2` when there is a branch statement in `B1` with a possible destination in `B2`. The CFG also contain a single entry node `entry` and a single exit node `exit`. The CFG tells us about what possible paths the program might execute when the function is executed.

1.2.4 Dominators

Given a CFG `G=(V,E)` with single entry node `entry` and single exit node `exit`, a node `D` is a *dominator* to node `N` in `V` if every path from `entry` to `N` must go through `D`.

A node `N` is a *postdominator* of `D` if any path from `D` to `exit` must go through `N`.

A node `N` is the *immediate postdominator* of `D` if `N` is the first postdominator on any path from `D` to `exit`.

1.2.5 Unreachable-Code Elimination

For any given function the CFG `G=(V,E)` with single entry node `entry` and single exit node `exit` the unreachable basic blocks are simply those which there are no path to from `entry`.

Unreachable code is code in basic blocks which we know for sure cannot be executed, the elimination of unreachable code does not necessarily have a direct effect on the execution speed of a program but is guaranteed to decrease the size of the program. It is however possible to get faster execution speed as the result of unreachable code elimination, the size of the function could for example get small enough to fit in the L1 instruction cache as the result of removing unreachable-code.

Another possible way to get a speed increase as the result of removing unreachable code is that we sometimes can use *straightening* [17] on the CFG.

Straightening is an optimization where we detect pairs of basic blocks `(B1,B2)` where `B1` only have `B2` as successor and `B2` only have `B1` as predecessor, this means that there is an unconditional jump from `B1` to `B2` and we can therefore replace the jump with `B2`.

2 Inlining heuristics in studied compilers

In this thesis we study the inliner in 3 different compilers.

2.1 IAR Inliner

The inlining is done in the global optimizer (middle-end) of the compiler which means that the inliner is back-end independent and works on an intermediate representation of the code. This also means that in general the inliner will not make any back-end specific decisions.

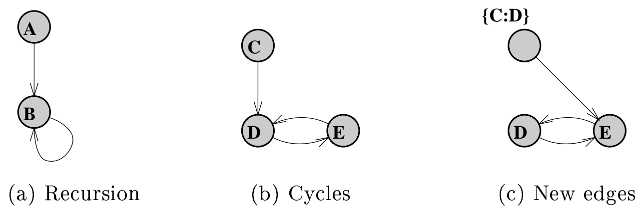
The current inlining heuristic is solely based on the expected size of a function. It is the expected size of a function since we are working with intermediate code but also because we do inlining in an early phase of the optimizer and later optimizations can affect the size.

The optimizer will start by building a call graph and go through the functions in reverse invocation order (i.e leafs and up, bottom-up), for every function that has callers the decision to inline or not is taken based on a combination of some of the properties mentioned in 1.1. If the inliner decides that the function should be inlined it is inlined at all call sites. There are two `#pragmas` that can be used to override the heuristics, one that forces a function to be inlined and one that prevents a function from being inlined.

There are some situations that can occur in a call graph where this inlining heuristic will refuse to inline. In Figure 1a there is a back-edge on `B` which means that `B` has a call to itself. This inliner will never inline recursive calls.

A program where the call graph has a cycle like in Figure 1b it's never possible that both edges in the cycle get inlined. The only possible inlining in the cycle is that either `D` inlines the call to `E`, or `E` inlines the call to `D`. If both edges are suitable for inlining we will inline the edge which we look at first.

When we inline a function which itself contains function calls we get new edges in the call graph, in Figure 1c inlining of `D` into `C` has resulted in a new edge from `C`. New edges that are the result of inlining are never considered for inlining in this inliner (i.e only the original call graph will be considered).



(a) Recursion (b) Cycles (c) New edges

Figure 1: Situations in call graph where the current ICC inliner refuse to inline

2.2 RealView Inliner

RealView software development tools are sold by ARM [1], although they have several software development tools what we're interested in for this thesis is of course their C and C++ compiler called `armcc`. According to the compiler documentation the inlining decision is ultimately based on a "complex decision tree" [9]. But as a general rule the following criteria are taken into consideration when deciding if a function should be inlined.

- The size of the function and how many times it is called, smaller functions are more likely to be inlined.
- Optimizations flags `-Otime` and optimization level in general.
- Large functions are normally not inlined.
- Reserved keywords that the programmer can use to increase the chances of a function being inlined.

The documentation describes a heuristic that sounds similar to the heuristic used in the IAR compiler. A guess is that they also choose to inline at all call sites or none, the fact that they mention the number of call sites in the criteria indicates a heuristic like this. Some experimenting with the inliner also indicated that RealView choose to inline in an all or none fashion.

2.3 GNU Compiler Collection (GCC) Inliner

The GCC compiler supports a number of different programming languages and targets. The compiler is open source and free to use under the GNU General Public License (GPL)[2]. In this thesis we use a GCC compiler with an ARM back-end provided by CodeSourcery [3].

This is by far the compiler studied in this thesis that has the most possibilities for the user to configure the inliner by using parameters. Inlining bounds such as, maximum growth of module, maximum depth when inlining recursive functions, maximum size of a function, etc. can all be set by using different options to the compiler. Typically a user would use the default values for all these parameters (perhaps not even aware about inlining at all), but it's important to know about this as.

This means that if a user thinks inlining expands the code too much, there's always the possibility to set the upper limit of growth explicitly and try again. Neither IAR nor RealView support this kind of work flow.

There is a document about the GCC call graph module and interprocedural optimizations [14] that talks about inlining in GCC. However since the document is from 2004 we decided to gather information about the inliner straight from the source code to make sure that we get an up to date view of the heuristics. The inliner in GCC works as follows, begin by inlining functions that are small enough that they should always be inlined (i.e call cost is higher or equal to inlining the function). After this GCC inlines functions that are considered "small enough", the upper bound for what is considered small or not can be set with parameters to the compiler. The calls to the "small enough" functions are

[2]http://www.gnu.org/licenses/gpl-3.0.txt

assigned a score that is based on the number of call sites to the function and if the call is nested, the call sites are then inserted into a heap sorted on this score. Call sites are then removed from the heap and inlined until it is either empty (i.e empty call graph, all possible inline candidates are inlined) or until an upper bound in total growth is met (this upper bound can also be set explicitly with parameters), this means that GCC can inline a subset of call sites to a function. After this a last pass over functions that only have one call site are considered. GCC also handles inlining of recursive functions in a way so that the function is inlined down to a given depth. GCC also supports inlining decisions based on profiling data from an earlier run of the application.

3 A new inliner

3.1 Ideas for a new inliner

We believe that the biggest problem with the current ICC heuristics is that it inlines too much, results (Figure 1 on page 22) in this thesis will also show that the current heuristics generates a large code growth. Since the current heuristics is solely based on an estimation for each function which is then compared to a given limit it means that the inliner will inline without any control over the total code growth. If there are many functions that pass the upper limit the growth will sometimes be very large.

The new heuristic include a more detailed analysis of the functions and call sites to be able to make more accurate inlining decisions and also prevent the code from growing out of control. To do this every call site will be assigned with a *badness* value which is calculated by a score function, here $badness(x) < badness(y)$ would mean that call site `x` is inlined before call site `y`. To prevent the code from growing too much we shall also keep a history of how earlier inlining decisions have affected the code growth.

3.1.1 Selective inlining

For the new heuristic where a call site is assigned a badness value it's obviously not a good idea to go for the old heuristic style with inlining all or nothing of a function since this would make it impossible to have a ranking among call sites to the same function. The new heuristic can therefore handle inlining of just a subset of the call sites to a given function, the reason for this is that some call sites are likely to be more suitable for inlining and these should not be ruled out because there are other calls to the same function that are less likely to be good inlining candidates.

As an example consider the pseudo-code in Listing 1, we have 5 different calls to a function `A`. With the old heuristics we would either inline all these calls or none but now that every call site has a corresponding badness value (rather than a badness value for the whole function) we have the possibility to have a ranking between these call sites.

Listing 1: Selective inlining

```
A()
A()
for
   for
      A()
A()
A()
```

This is one example why we should support selective inlining. It is however important to remember that the number of call sites to a function should still be part of the evaluation, but it should not be enough to rule out a specific call site for inlining.

3.1.2 Heuristics for call sites

We want the inliner to be able to make some kind of estimation of how likely it is that a call site is used often. Define the $hotness(X)$ of a call site `X` so that $hotness(X) > hotness(Y)$ means that it's more likely that call site `X` is executed more often than `Y`. Some compilers (GCC for example) have support for using profiling data as input to the inliner, the profiling information can then be used to give a good approximation of which calls are hot and which are not.

Currently we do not have the possibility for such feedback from the IAR Embedded Workbench Profiler but we still would like the inliner to give call sites that are likely to be hot a little boost. To do this we use a very simple analysis that checks the loop-depth of a call site. Where a higher loop-depth would increase hotness.

With the addition of code hotness analysis to the inlining heuristics we should be able to rank the nested call to `A` in Figure 1 higher than the other calls to `A`, and perhaps only choose to inline this call. Without selective inlining this kind of call site boosting would not be possible.

3.1.3 Unreachable-Code Elimination

The idea of predicting further optimizations on functions as a results of inlining for was first described by J. Eugene Ball in a research article [12]. We can think of this as the results of a constant propagation made over function calls which results in the possibility to do unreachable code elimination.

Every function is analyzed for parameter dependency and then annotated with information about which parameters and what possible savings are possible if the parameters were replaced by constants. Consider the program in Listing 2.

Listing 2: Unreachable code elimination possibilities after inlining.

```
1  void foo(int x)
2  {
3     if(x == 1) { ... }
4     else { ... }
5  }
6
7  void bar()
8  {
9     foo(randomNumber());
10    foo(1);
11 }
```

The function `foo` have a dependency on the parameter `x` because if we know the value of `x` the optimizer can remove the conditional statement and replace the if/else branches with the appropriate branch depending on the value of `x`. The function `foo` is therefore annotated with information about the possibility to remove unreachable code if the function is inlined at a call site with a parameter that at compile time has a known constant. The function `bar` that is a caller of `foo` can then use this information when considering inlining of the calls.

The call to `foo` at line 9 has an unknown parameter value at compile time, therefore we must expect that we need a copy of the full function `foo` if we inline at this call site. When considering the call at line 10 the inliner should be able use the information annotated with `foo` and realize that all code in the else branch will be removed if the call is inlined. The inliner can therefore assign the second call a lower badness value compared to the first call.

3.1.4 Termination of inlining

There are a number of different ways for an inliner to terminate. With the current ICC inliner we terminate when all callees in the call graph has been examined, but with the new inliner we have other possibilities. Since the new inliner is going to keep a badness value for all edges in the call graph one way to terminate would be to inline until the currently lowest badness-value are above a certain limit. Another way to terminate is that the inliner runs until a certain limit in growth is met. The new inliner will terminate on one of the following criteria.

1. No more edges in the call graph.
2. The next edge to inline has a badness value higher than a given limit.
3. The next edge to inline would result in a growth larger than a given limit.

4 Implementing the new inliner

4.1 Analyzing functions for parameter dependency

We need to annotate every function with information about the possible savings that can be achieved if the functions is called with some constant parameters. We use this information in a later stage when assigning the badness value to edges in the call graph. To do this we scan the function and search for conditional statements (if/else/switch) and find out if any of the conditions are fully dependent on any of the formal parameters to the function. What we are looking for are conditions that can be composed as follows:

1. The condition is composed only of parameters and constants, the parameters must not have been redefined before the condition.
2. The condition is composed of local variables that are defined by formals and constants (again no redefinition of the formals before the definition of the local).

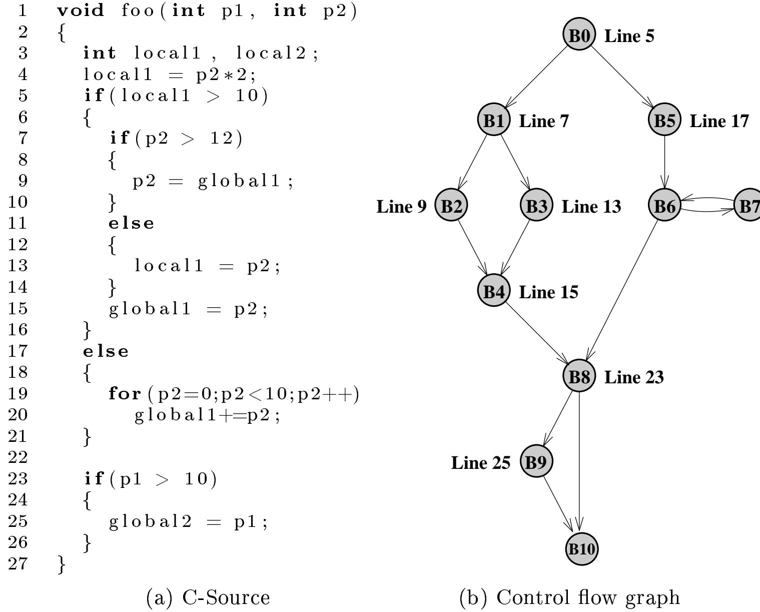
When a condition like this is discovered we know that if constants are available at compile time we can at least evaluate the condition and remove the conditional statement, it's also possible that we will be able do remove dead code as a result of the elimination of the the condition. The condition will be saved along with the function with information about what parameters it is dependent on. Since it's possible that the condition is dependant on more than just one parameter we save a bit-vector where the dependant parameters are set and parameters that's not used in the statement are cleared. The algorithm in Algorithm 1 is used to find out which basic blocks are unreachable if we know what the condition evaluates to (true or false).

Algorithm 1 Calculate unreachable basic blocks

```
1 PD = ImmediatePostDominator(T)
2 All = BFS(T, PD)
3 TD = TrueBranch(T)
4 FD = FalseBranch(T)
5 UnreachableIfTrue = All-BFS(TD, PD)
6 UnreachableIfFalse = All-BFS(FD, PD)
```

From the node `T` which includes the conditional statement find out the post dominating node `PD`, do a Breadth-first search (BFS) from `T` to `PD`. The basic blocks found during this BFS are all reachable if we don't know the result of the condition.

Now assume that the condition is true and collect all the basic blocks reachable from the block we jump to, to the post dominating block. After this do the same thing but assume that the condition is false. To find out if some nodes are no longer reachable if the condition is true just subtract the reachable nodes in the true-branch from the all-nodes. Then do the same for the false-branch. The sets with unreachable blocks are then saved together with the node `T` for later use.



```c
void foo(int p1, int p2)
{
  int local1, local2;
  local1 = p2*2;
  if(local1 > 10)
  {
    if(p2 > 12)
    {
      p2 = global1;
    }
    else
    {
      local1 = p2;
    }
    global1 = p2;
  }
  else
  {
    for(p2=0;p2<10;p2++)
      global1+=p2;
  }

  if(p1 > 10)
  {
    global2 = p1;
  }
}
```


(a) C-Source (b) Control flow graph

Figure 2: Analysing a function for constant dependencies

Consider the example in Figure 2. Assume that the left path in Figure 2b is taken if the condition is false.

The first conditional statement discovered is the conditional statement at line 5 which correspond to the basic block `B0`. The condition is examined and we detect that the condition is dependent on a local variable `local1` and a constant `10`. We then have a look at the definition of `local1` and find out that it is defined by the parameter `p2` and a constant `2`. This means that the condition in `B0` is fully depend ant on the parameter `p2` so this is a candidate for test elimination and unreachable code removal. In this situation we save the actual condition (call this `Test1`) along with enough information so that we can map variables used in the expression to parameters and then we run the algorithm described in Algorithm 1. The result of Algorithm 1 will be `PD=B8`, `TD=B5`, `FD=B1` and the sets `All={B1,B2,B3,B4,B5,B6,B7}`, `UnreachableIfTrue={B5,B6,B7}`,
`UnreachableIfFalse={B1,B2,B3,B4}`.

The sets `UnreachableIfTrue` and `UnreachableIfFalse` will now be saved together with `Test1`. There will also be a `Test2` and `Test3` saved with this function. Where `Test2` is the conditional statement at line 9 and `Test3` is the conditional statement at line 23. The function have then been analyzed for constant parameter dependencies and we will be able to calculate possible savings if the function is inlined at call sites with known constant parameters.

4.2 Building the edge queue

Given a call graph for the program we construct a minimum priority queue which hold all edges in the call graph. The edges in the queue will be keyed on

the badness value. For every edge in the call graph that is an inline candidate the following properties is used to calculate a badness value.

- The loop depth of this call site.
- Constant parameters at this call site.
- The size of the callee (expected).
- Number of calls to the callee.
- Does the callee have the inline keyword.
- Recursion penalty (optional) see chapter 4.4.

If the callee is tagged with possible dead code elimination and the call site have constants available for these parameters the size of the callee will be estimated to a new smaller size depending on how much we expect to save.

4.3 Inline driver

The inline driver is where the actual inlining takes place, this is where we pick edges from the queue created in the analysis part and inline them. The inline driver described in this chapter is heavily inspired by the inline analysis driver presented in research article about inlining [13] written by D. Chakrabarti et al. As described by Chakrabarti we sometimes need to update some of the remaining edges in the queue after we inline a function, we used the same method as proposed in the article and keep a look up table to validate the edges we extract from the queue.

The pseudo code for the inline driver is pictured in Algorithm 2. For the moment just assume there is an algorithm `UPDATE` which given an edge that has been inlined, the queue, the look up table and the current call graph will update all necessary data structures. How the `UPDATE` algorithm works is described in the next section.

Algorithm 2 Inline driver

```
GET_BEST(Q, L)
  valid = false
  while !valid:
    e = POP(Q)
    if L.exists(e.key) and L[e.key].badness == e.badness:
      valid = true
    if Q.empty:
      return null
  return e

DRIVER(Q, CG)
  L <- empty
  growth = 0
  while growth < allowedgrowth
    edge = GET_BEST(Q,L)
    if edge == null or edge.score > badnesslimit
      return
    growth += INLINE(e)
    (CG, Q, L) = UPDATE(e, Q, L, CG)
```

4.3.1 Updating data structures after inlining of one call site

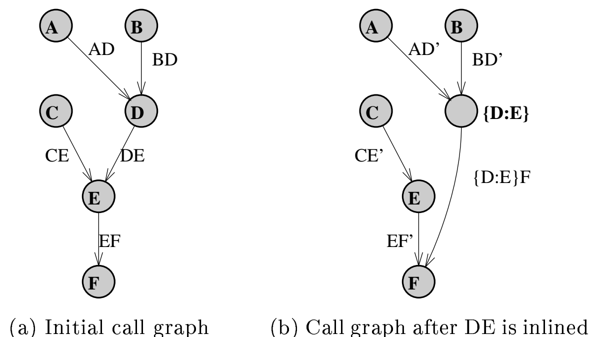
After we inline a call site the call graph change and we need to calculate new badness values. We could rebuild the entire call graph and calculate new badness values for all the edges in the new call graph just like we did in the initialization phase but this would not be efficient and it's not necessary to recalculate all badness values.

Algorithm 3 describes the dependencies and necessary updates we need to do after inlining a call site.

Algorithm 3 Updating call graph and edges after inlining

```
1  UPDATE(e, Q, L, CG)
2    caller = e.Caller
3    callee = e.Callee
4    CG = RemoveEdge(e, CG)
5    for all c in callee.Callees:
6      newedge = NewEdge(caller, c)
7      CG = AddEdge(newedge, CG)
8      Q.push(newedge)
9      updatededge = NewEdge(callee,c)
10     Q.push(updatededge)
11     L[updatededge.key] = updatededge
12     for all c2 in c.Callers:
13       updatededge = NewEdge(c2, c)
14       Q.push(updatededge)
15       L[updatededge.key] = updatededge
16   for all c in callee.Callers:
17     updatededge = NewEdge(c, callee)
18     Q.push(updatededge)
19     L[updatededge.key] = updatededge
20   for all c in caller.Callers:
21     updatededge = NewEdge(c, caller)
22     Q.push(updatededge)
23     L[updatededge.key] = updatededge
24   return (CG, Q, L)
```

To understand what the different parts of Algorithm 3 is needed for consider the call graph given in Figure 3a as part of a much larger call graph, we assume that the edge with the currently lowest badness value is the edge `DE` and that the inline driver have inlined the `DE` edge and now calls `UPDATE` to make sure that dependent edges are updated. The result of inlining DE is shown in Figure 3b.



(a) Initial call graph (b) Call graph after DE is inlined

Figure 3: Demonstration of Algorithm 3

We begin by removing the edge that got inlined by calling `RemoveEdge` on line 5.

After this the first for-loop updates the edges going out from `E` which in this example results in a new edge `EF'` , there is also a new edge `{D:E}F` which is added both to the call graph and to the queue. The nested for loop on line 12 makes sure that all other edges to `F` would get updated too. The reason we need to update all the old edges to `F` is because `F` now has one more caller.

The for-loop that starts on line 16 updates edges going in to `E`. In this example this means that the edge `CE` is updated and becomes `CE'`. We need to update this edge because there are fewer callers to `E` now. The last for-loop updates callers to caller. In the call graph we can see that `A` and `B` are callers of `{D:E}` and these edges should therefore be updated since it's likely that the size of `{D:E}` is larger (or at least different) than `D`.

For all edges that are updated we save the new edge in the look up table `L` to make sure that we extract valid edges from `Q` the next iteration in the inline driver.

4.3.2 User interaction

The badness and maximum growth limits are given as parameters to the compiler, we have tried the inliner with many different combinations of these two limits (see Appendix A).

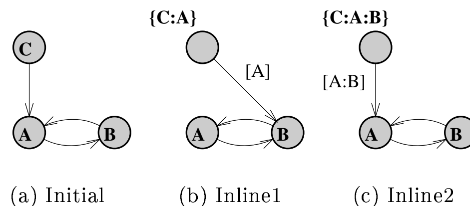
4.4 Avoid getting stuck in local parts of the call graph

Consider the call-graph in Figure 4 and think of it as part of a larger call-graph. Assume that currently the edge with the lowest badness value is the one between `C` and `A` so the inliner choose to inline this edge which will result in the call-graph shown in Figure 4b.

The inliner then choose to inline the edge between `{C:A}` and `B` which results in the call-graph shown in Figure 4c, this call-graph looks exactly like the one we started with. If we don't have any history of earlier inlining decisions this means that the inliner would inline the same edges again and again until it stops because of code growth, thus it's stuck in a local maximum here and no other edges outside this part of the call-graphs will get inlined.

To prevent this from happening we annotate edges that are the result of earlier inlining. In Figure 4b the edge between {C:A} and B is annotated with [A] because it is the results of inlining A into C and in Figure 4c the edge between {C:A:B} and A is annotated with both [A,B]. The inliner will penalize edges where it have already been, this means the the annotated edge in Figure 4b is not penalized, but the annotated edge in Figure 4c is penalized. The penalty depends on the nested depth of the edge and penalties are accumulated.

It's still possible for the inliner to inline in the same order again, but for every time it choose these edges the new edges created will get a larger penalty. By annotating the edges we make it possible for the badness calculation to take these situations into consideration and penalize edges so we don't get stuck in a local maximum of the call graph.



(a) Initial (b) Inline1 (c) Inline2

Figure 4: Annotating edges to avoid local maximum in a pseudo-cycle.

5 Test suite

5.1 Applications

To find out how the inliner performs on real code we use a number of different test applications where we are interested in size and speed with and without inlining. Inlining will not be made across modules unless otherwise is stated.

We want to test the inliners on rather large applications since it's more likely that the inliner will find candidates for inlining then. However it is not as easy as it may sound to find a set of applications to compile and run, the ICC compiler only supports a subset of C++ called Embedded C++ (EC++) which means that we cannot just take a random open source C++ application and compile it. It can be hard enough to find applications to compile when you only have to consider using one compiler and in this thesis we have to make sure it compiles with 3 different compilers.

5.1.1 Dinkum

The Dinkum C++ proofer [5] test suite is used to test for C++ standard compliance, the source code is therefore not like typical application code. The structure of the different tests are basically the same, one or more "paragraphs" in the C++ standard is tested and this means that one or more "sub tests" for every tests needs to be done. Basically what we're doing here is unit testing paragraphs in the ANSI standard.

For each sub test made a "check" function is called to make sure that the result is what we expected, this means that if a test file includes a lot of sub

tests then there will be a lot of calls to these check functions. Even though this test suite does not represent typical application code it is interesting in this thesis anyway because the Dinkum test suite will make use of a huge amount of STL code and the results will show that the current IAR inliner have a lot of problems with some of the constructs in these tests.

The result presented for Dinkum is all tests put together, a total of 97 C++ source files.

5.1.2 Open Physics Abstraction Layer (OPAL)

OPAL[7] is an open source low-level physics engines written in C++ that can be used in games, robotics simulations, etc. We will benchmark the test suite that comes with the source code package.

5.1.3 Persistence of Visions Ray-tracer (POV-Ray)

An open source ray-tracer called POV-Ray [8], we will use the included benchmark for POV-Ray to do our benchmarking. Because it takes a lot of time to run this benchmark we use the Freescale hardware mentioned in section 5.1.3 when running this application.

5.1.4 Turing Machine Simulator

An open source Turing machine simulator [11] written in C++, uses a lot of STL code.

5.1.5 SPEC2000 CPU

We will use 6 of the applications in the SPEC2000 [10] integer benchmark suite, there is a total number of 12 applications that use C or C++ in this suite but because we didn't get a copy of SPEC until very late we only had time to patch[3] the applications that were the easiest to get through the compiler.

It's important to note that the results from SPEC presented in this thesis is not SPEC benchmarks, we will only use the source code and test data from the SPEC benchmark suite and never intend to get actual SPEC benchmark scores (to get real benchmarks would require too much work).

We also tried the Multi-File-Compilation (MFC) feature of the IAR compiler on some of the applications which means that the inliner is able to do cross module inlining.

5.2 What we measure

The difference in size when enabling inlining is of course interesting. When we talk about size in this thesis we refer to the code segment (.text) of the output file. We expect none or very small changes to the data segment when enabling inlining and if the data segment was to be included in the size it's possible that a large code segment increase would “drown” in the data segment when we present relative differences.

[3] None of the applications could be compiled out of the box. A typical patch is argc/argv parameter support

It is possible that enabling inlining will change the amount of library usage. To demonstrate this consider the example in Listing 3.

Listing 3: Example where inlining would reduce the amount of library usage

```
extern void A();
extern void B();

static void foo(int x)
{
  if(x == 0)
    A();
  else
    B();
}

static void bar()
{
  foo(0);
}
```

Assume that `A` and `B` are defined in a library, there are no other references to `A` or `B` other than the one shown and there are no other calls to foo other than the one shown. If we do not allow inlining the linker[4] will have to include both `A` and `B` in the executable since there are references to both `A` and `B` from `foo`. However if inlining is enabled and `foo` is inlined to `bar` further optimizations will detect that the if-statement is always true and the else part that include the call to `B` will be unreachable and therefore removed.

After inlining `foo` into `bar` we will also remove the only reference to `foo` and the linker can therefore remove `foo` and there will be no more references to `B`, the linker will therefore only include the function `A` from the library.

Because the inlining can affect the amount of library usage we will remove the overhead of library code usage in the base configuration before comparing size differences.

Compilers for C and C++ typically come with their own implementation of the standard libraries[5]. The applications we test the inliners on will of course use the standard libraries available for C and C++ and to make the comparisons more fair we choose to use the same standard library for all compilers. This means that all compilers will be faced with the same input and get the same possibilities to inline, no `#pragmas` or other compiler specific keywords will be used in the source code we benchmark.

[4]With linker we mean the IAR Systems ILINK linker or any other linker that perform similar optimizations.

[5]All compilers studied in this thesis comes with a different implementation of the standard libraries

6 Test framework

6.1 Tools and target

The target we build for is an ARM1136JF-S processor core and even though there is a VFP[6] for floating point operations in this processor we will compile for software floating point because the simulator had problems simulating the VFP.

6.1.1 Simulator

We have used the ARM Symbolic Debugger [2](`armsd`) to measure execution time. The number of clock cycles used from the entry of `main` until `main` returns is considered the run time of the application.

6.1.2 Hardware

When it takes too long to run an application in the simulator we instead use real hardware, we have used a Freescale i.MX31 [6] application processor and Embedded Linux as operating system. Running applications under a real operating system poses some problems with the C++ library that is delivered with IAR compilers. The library does only support semihosted i/o. Semihosted means that to do i/o (printing, reading/writing a file, etc) you write an i/o code to one register and an address to store the result in another register and then do a software interrupt. The simulator running the application will then read these registers and to appropriate operations using the hosts i/o.

This not possible when running on real hardware with Embedded Linux. We have to replace the software interrupts with kernel systems calls which means that we get an interrupt and cache-flushing when doing i/o. We believe that this overhead will have very little effect on the benchmarks. Time measure is done using the standard UNIX command `time`.

[6]VFP is a coprocessor extension to ARM that provides floating-point computation support.

7 Measurements

7.1 Intermediate code vs generated code

In chapter 2.1 we said that the current heuristics in ICC is based on the expected growth of a function because the optimizer works with intermediate code. To see if it is reasonable for the optimizer to assume that the relationship between intermediate code and generated code is proportional we compiled 555 different C and C++ source files and the result is shown in Figure 5.

The leftmost point which deviates from the "line" is the file `graphic.c` in the SPEC benchmark vpr. When this application is built without graphics support all function bodies in this file are empty and the intermediate code size calculation is therefore close to zero. The size presented is module by module and it is possible that we would get more outliers if we instead would count every function for itself.

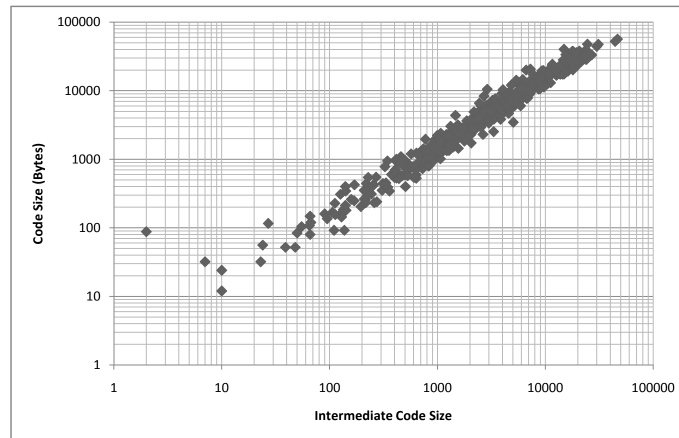


Figure 5: Intermediate Code/Generated Code relationship

7.2 Compilers

IAR is represented with the `IAR C/C++ V5.20` compiler for ARM with the optimization flags `-Ohs --no_inline` for the base configuration which use no inlining and `-Ohs` to use the current inlining heuristics. We also added a new option to the old heuristics where we only inline the functions in category 6 (see chapter 1.1). This configuration are called "IAR Smaller".

We have used the `ARM C/C++ RVCT3.1` compiler from RealView with the optimizations flags `-O3 -Otime --no_inline` for the base configuration and `-O3 -Otime` for the configuration with inlining.

The GCC compiler we used is provided by a software consulting group called CodeSourcery [3] that provides an ARM tool chain based on GNU tools. We have used the `arm-2008q3` tool chain [4] which is based on GCC 4.3.2. The optimization flags for the base configuration are `-O3 -fno-inline` and the configuration with inlining is compiled with `-O3`.

7.3 Results

The code growth difference in % when using inlining compared to a base configuration with no inlining is shown in Table 1 and the difference in execution time is shown in Table 2. IAR New is represented in the tables by the configuration that use the least number of executions cycles for that particular application. All results for the new inliner are presented in Appendix A.

The "IAR Smaller" configuration produce good results with C++ code where we get both faster and smaller results. For C code this inliner does not have as much effect as with the C++ code, there is even one application (parser/MFC) where we get in increase in code growth and another application (`vortex`) where execution used more cycles compared to the base configuration.

The current IAR inliner produce faster code for all applications tested except for `vpr` where a 10% increase in code size also lead to 1% longer execution time.

The new IAR inliner produced faster code than the old inliner for all applications except for `parser` and `parser/MFC`, especially worth noting is `dinkum` where the new inliner reduced the number of cycles used by 38% at no cost in code growth.

GCC was the compiler that generated the largest code growth on average, to it's defense we must remember that it's possible to constrain the amount of code growth with GCC. One of the applications (`parser`) got a code growth of 117%. It's likely that much of this growth can be contributed to recursive functions which the other inliners rejected. Unfortunately it was not possible to run this application so we could not get any feedback on if the inlining of recursive functions had a big impact on run-time. There were only one application where inlining had a negative impact on execution time.

The RealView inliner seems to be very restrictive with inlining, always keeping the code growth under 6%. One important observation regarding inlining with RealView is that no application got a faster execution time compared to the base configuration with no inlining. Instead 3 of the applications took longer time to execute when using inlining. One possible explanation to the low amount of inlining in RealView is that they might have annotated their own libraries with keywords and pragmas to guide the inliner, and without this help the inliner is very defensive.

Application	Language	GCC	RealView	IAR	IAR Smaller	IAR New
dinkum	C++	19	5	24	-6	0
opal	C++	35	6	36	-7	26
POV-Ray	C++	34	0	29	0	12
Turing	C++	33	6	19	-22	4
Average	C++	30	4	27	-9	11
crafty	C	4	0	2	0	4
mcf	C	28	0	-3	-2	-1
mcf/MFC	C	-	-	-3	-2	-1
parser	C	117	0	15	0	10
parser/MFC	C	-	-	31	1	11
twolf	C	4	3	1	0	0
vortex	C	3	2	0	0	0
vpr	C	9	6	10	-1	0
vpr/MFC	C	-	-	14	-9	-9
Average	C	24	2	7	-1	2

Table 1: Code size % increase over a configuration without inlining (lower is better)

Application	Language	GCC	RealView	IAR	IAR Smaller	IAR New
dinkum	C++	37	-2	35	12	38
opal	C++	-	-	1	0	2
POV-Ray	C++	-	-	0	1	1
Turing	C++	27	-3	28	20	28
Average	C++	-	-	16	8	17
crafty	C	2	0	2	0	2
mcf	C	2	0	12	1	13
mcf/MFC	C	-	-	12	1	13
parser	C	-	0	5	0	3
parser/MFC	C	-	-	7	0	5
twolf	C	-	0	0	0	0
vortex	C	0	0	0	-4	0
vpr	C	-2	-1	-1	0	2
vpr/MFC	C	-	-	2	1	3
Average	C	-	-	4	0	5

Table 2: Run-time improvement % over a base configuration without inlining (higher is better)

8 Conclusions and future work

By looking at the results we can see that inlining is an optimization that has a much larger impact on C++ code compared to C code, regarding both the difference in size and speed. It is no surprise that inlining affects C++ code more than C code since an object oriented language encourages decomposition of code into smaller functions and classes which later can be reused by other objects.

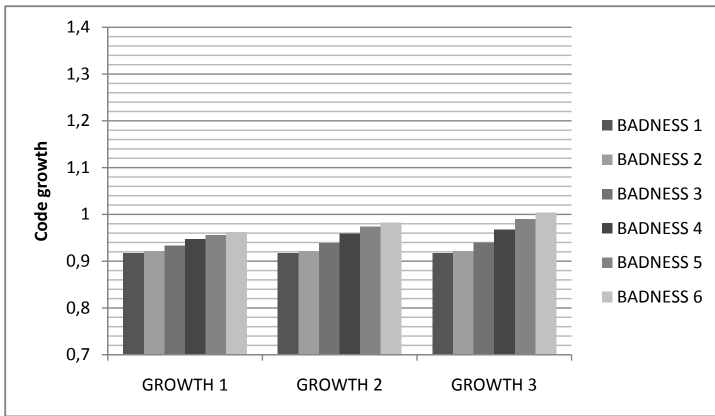
The current IAR inliner generates a large code growth just as we had expected, even though it usually means that we reduce the execution time we can draw the conclusion that the current inliner generates an unnecessarily large growth. We can for example compare the current inliner with the new inliner and see that for most of the applications the new inliner produces at least as fast code as the current inliner but at a much lower code growth. We believe that this is the result of an combination of inlining unnecessary function which does not increase performance and too much inlining which leads to more resource conflicts. The "IAR Smaller" configuration seems to be a good alternative for C++ code, it's easy to implement and the results are on average good. It also shows that inlining is a necessary optimization to use when optimizing C++ code.

The new IAR inliner seems to work good in most cases but it's not possible to find one single configuration which works best. It would be interesting to add more properties to the heuristics and then try running the inliner with different parts of the heuristics turned off. Examples of new properties are: current size of caller, number of formal parameters to callee, estimation of register pressure at call site and estimation of register pressure in callee, cache awareness (keep function small enough to fit in level 1 cache etc.). It would also be interesting to study the behavior of the heuristics when different properties are turned off.

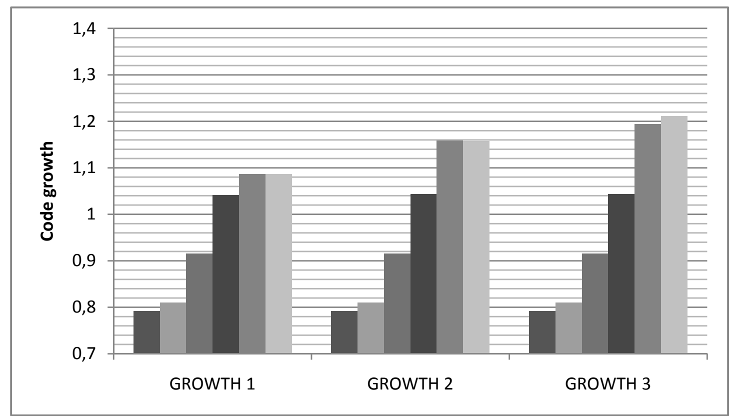
A Appendix

Every application has been compiled with the new inliner using 6 different badness combined with 3 differerent growth limits. In the charts presented in this appendix a higher number suffix on badness/growth means that the limit is higher (i.e badness 2 means a higher badness limit than badness 1).

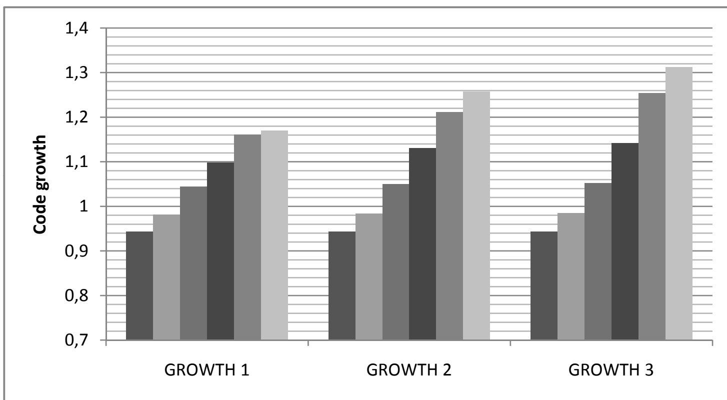
A.1 IAR New Inliner Code growth



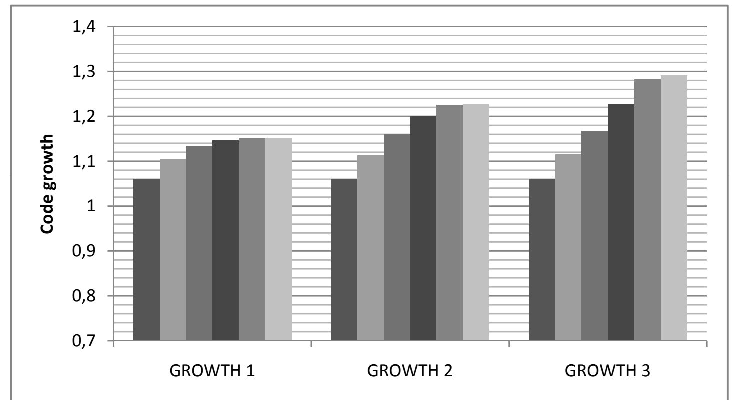
(a) Dinkum



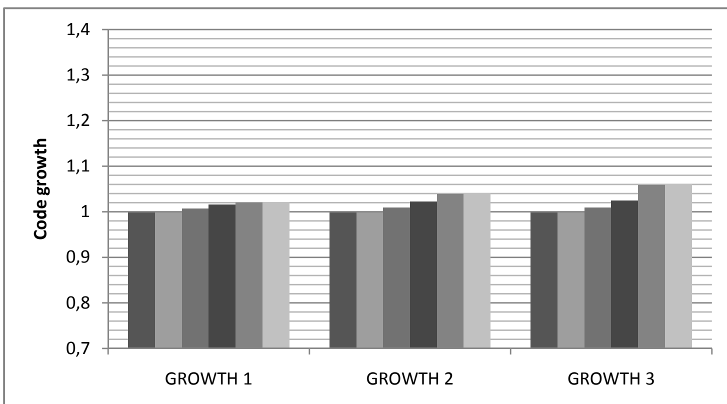
(b) Turing



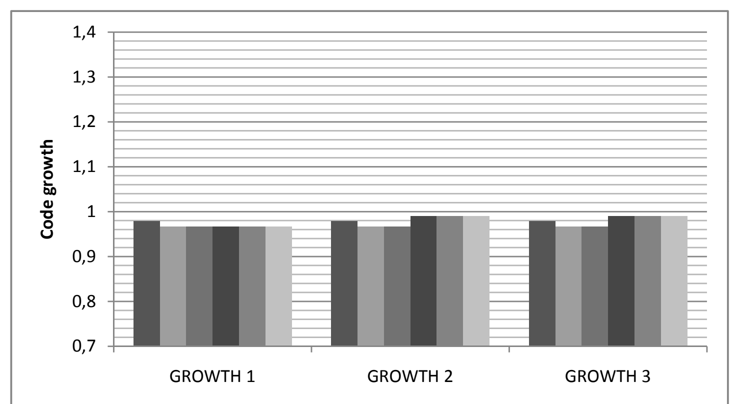
(c) Opal



(d) POV-Ray

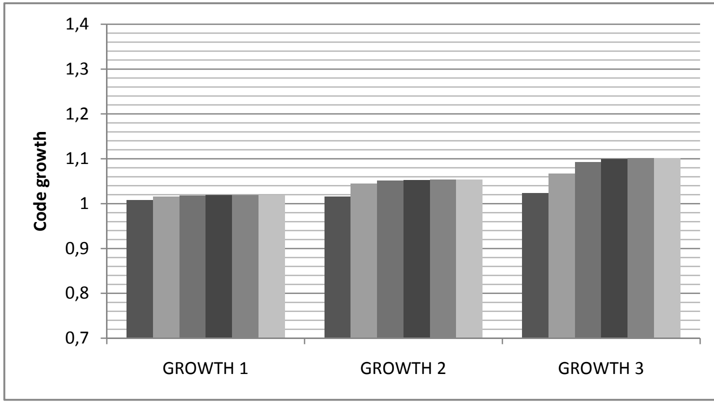


(e) crafty

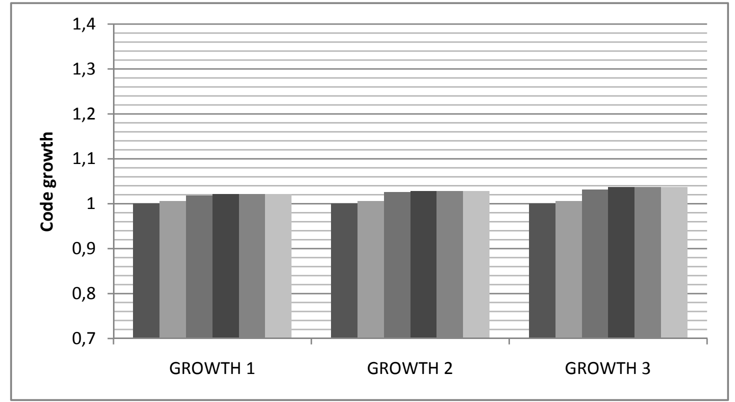


(f) mcf

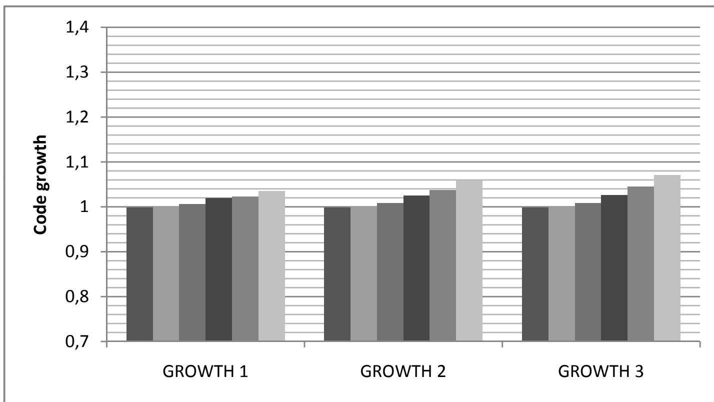
Figure 6: Code growth



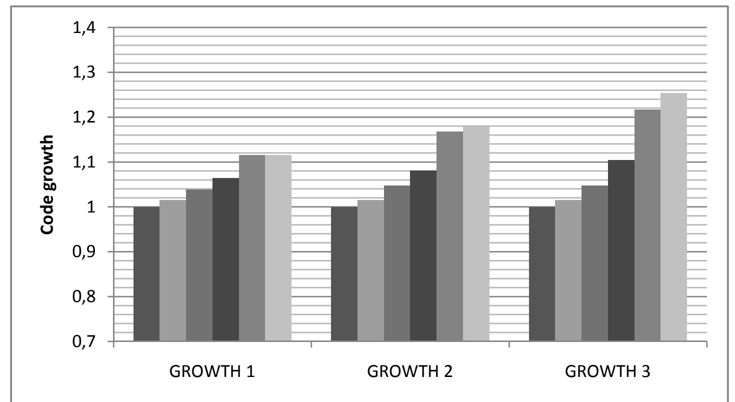
(a) parser



(b) twolf



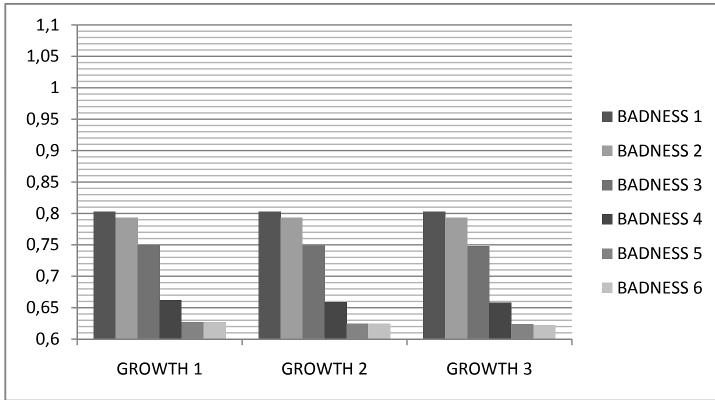
(c) vortex



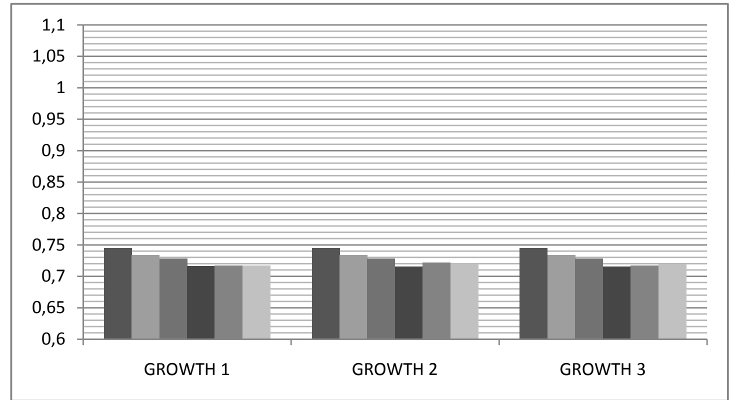
(d) vpr

Figure 7: Code growth

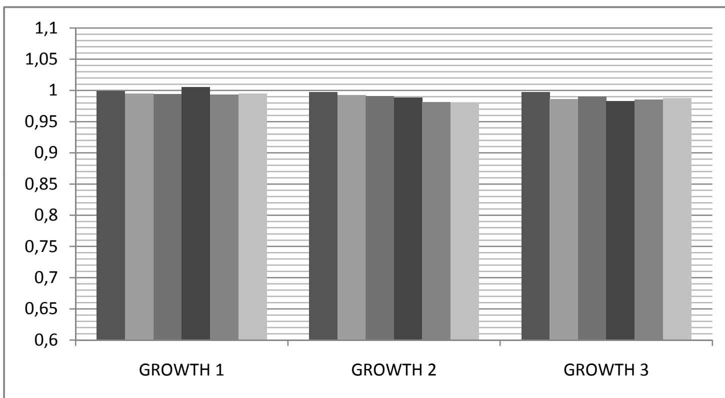
A.2 IAR New Inliner Cycles used



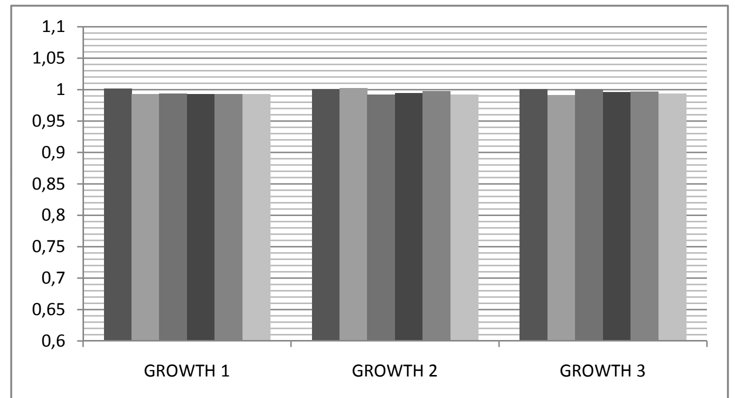
(a) Dinkum



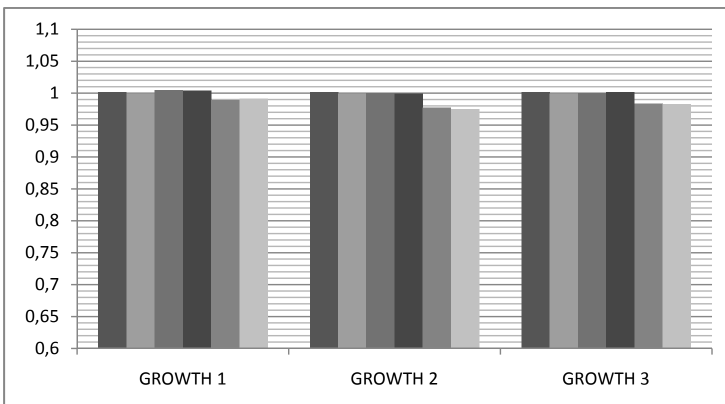
(b) Turing



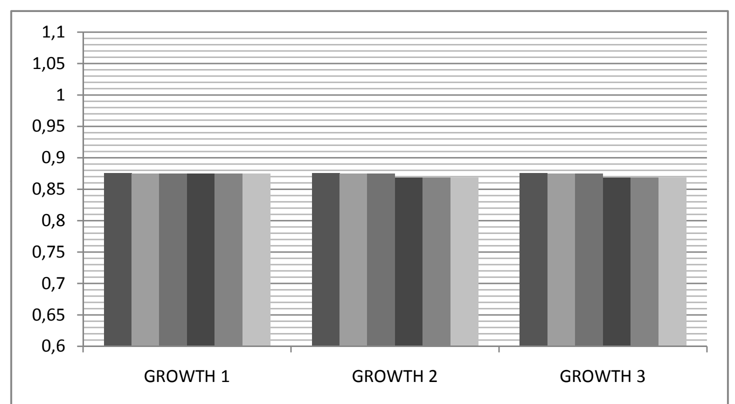
(c) Opal



(d) POV-Ray

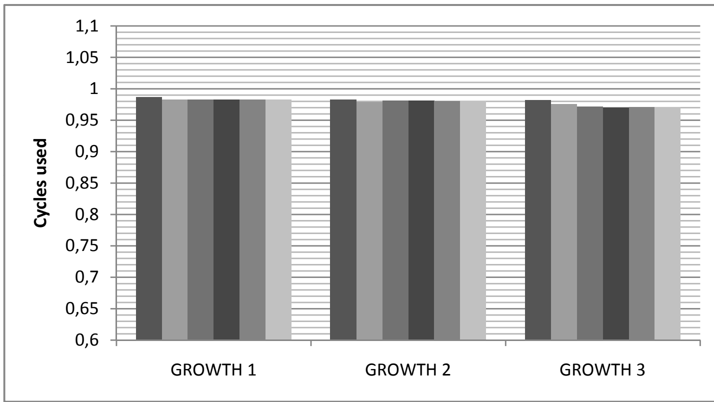


(e) crafty

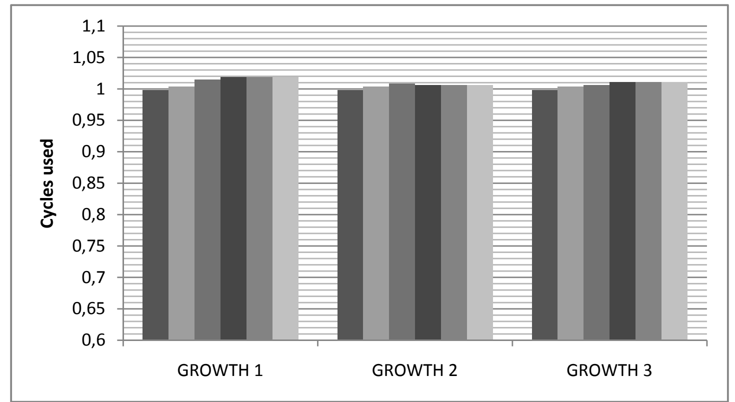


(f) mcf

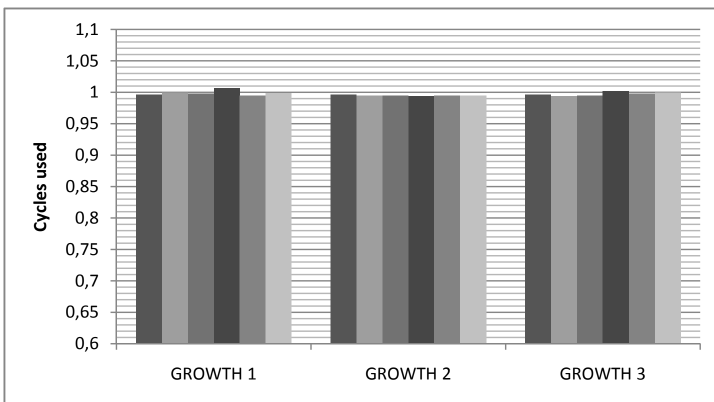
Figure 8: Cycles used



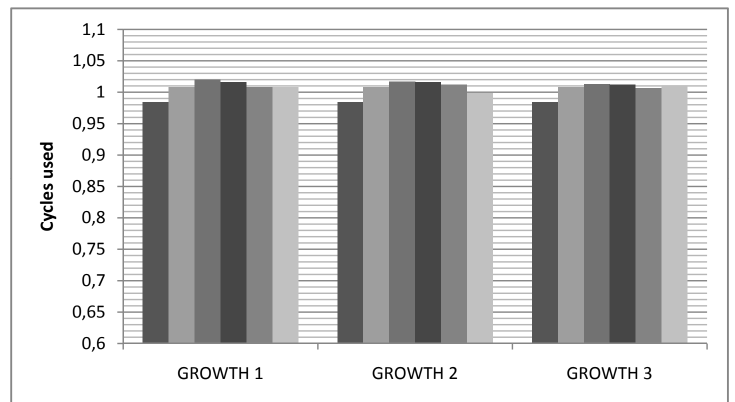
(a) parser



(b) twolf



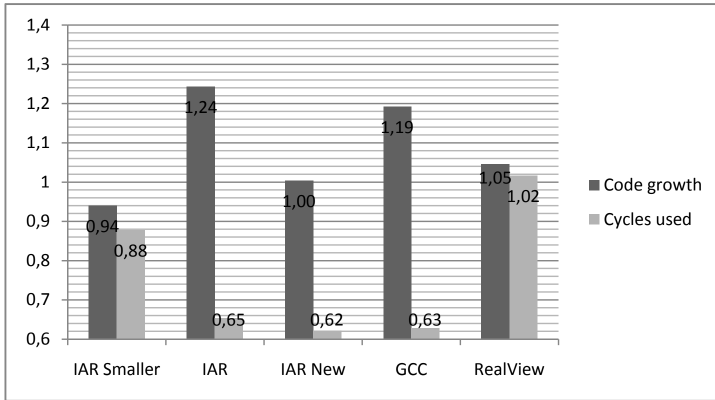
(c) vortex



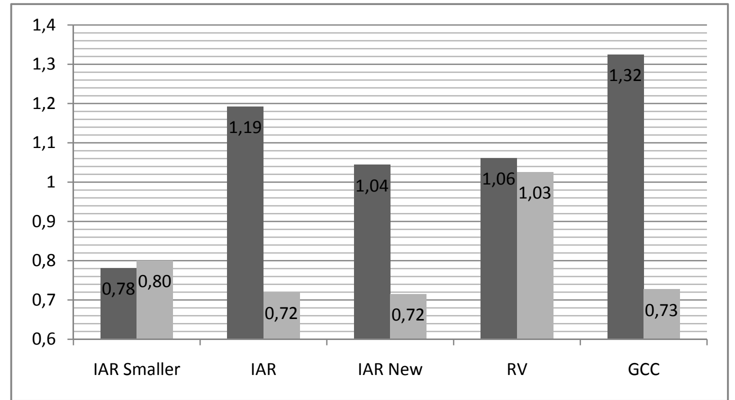
(d) vpr

Figure 9: Cycles used

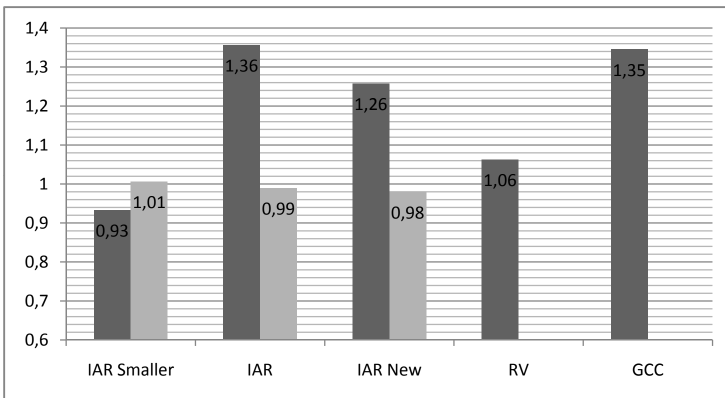
A.3 All inliners together



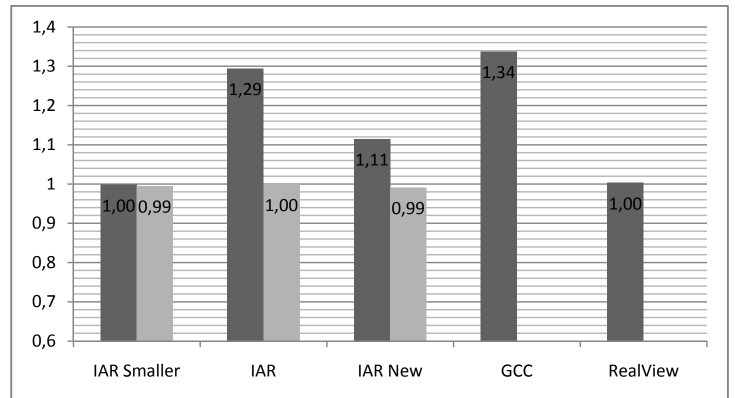
(a) Dinkum



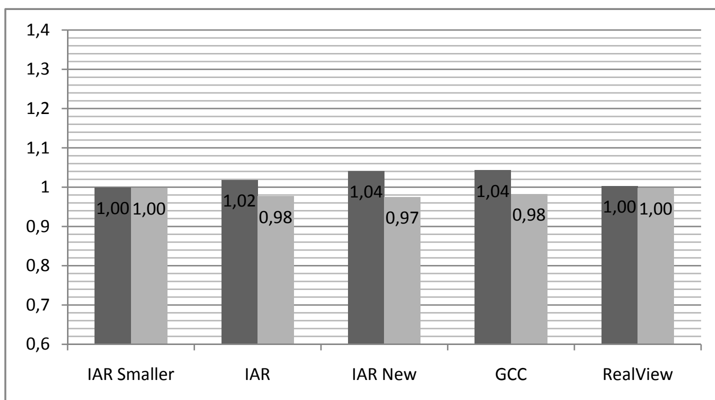
(b) Turing



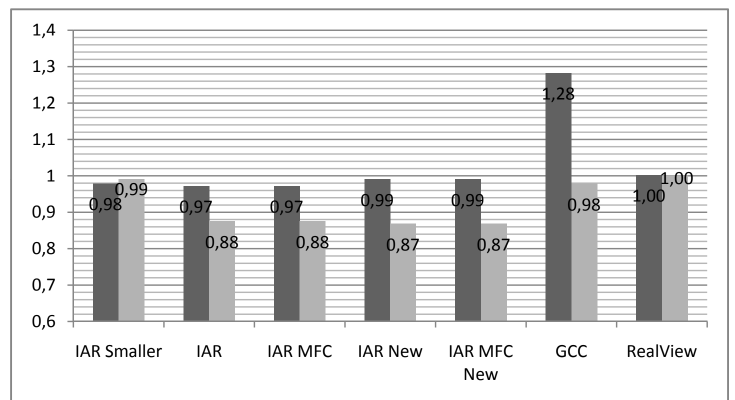
(c) Opal



(d) POV-Ray

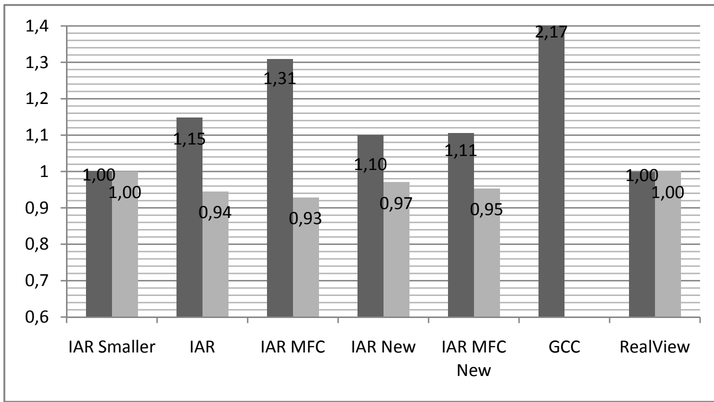


(e) crafty

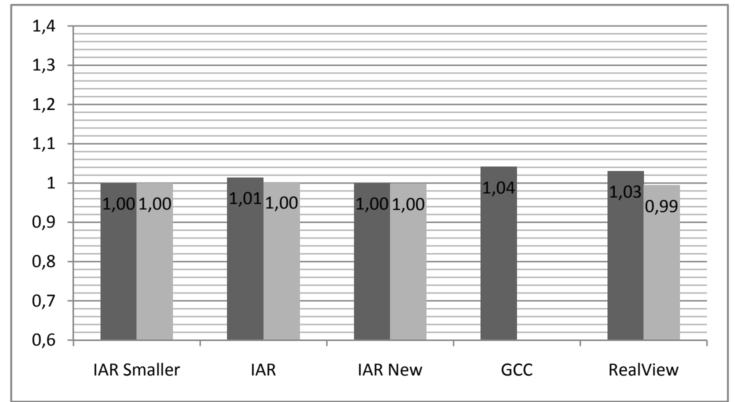


(f) mcf

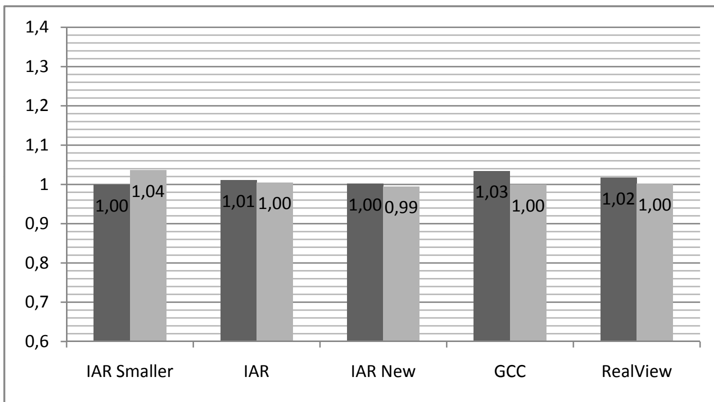
Figure 10: All inliners



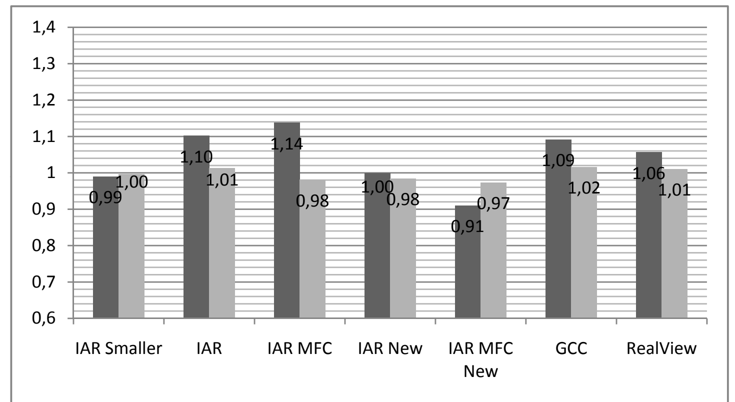
(a) parser



(b) twolf



(c) vortex



(d) vpr

Figure 11: All inliners

References

[1] ARM. The world leading semiconductor intellectual property (IP) supplier.
http://www.arm.com

[2] ARMSD. Simulator and Debugger for ARM code.
http://www.arm.com/products/DevTools/ARMSymbolicDebugger.html

[3] CodeSourcery. Software tools vendor.
http://www.codesourcery.com

[4] CodeSourcery ARM toolchain. Providers of ARM back-end for GCC.
http://www.codesourcery.com/gnu_toolchains/arm/

[5] Dinkumware. The makers of the STL library used in ICC.
http://www.dinkumware.com

[6] Freescale i.MX31 application processor. Semiconductor manufacturer.
http://www.freescale.com/imx31/

[7] The open physics abstraction layer. A high-level C++ interface for low-level physics.
http://opal.sourceforge.net

[8] The persistence of vision raytracer. An open source raytracer written in C++.
http://www.povray.org

[9] RealView Compilation Tools. Compiler User Guide Version 3.1.
Manual for RealView compiler.
http://infocenter.arm.com/help/topic/
com.arm.doc.dui0205h/DUI0205H_rvct_compiler_user_guide.pdf

[10] Standard performance evaluation corporation. Non-profit corporation that maintain a standardized set of benchmarks.
http://www.spec.org

[11] Turing machine simulator. An open source C++ implementation of a Turing machine.
http://sourceforge.net/projects/turing-machine/

[12] J. Eugene Ball. Predicting the effects of optimization on a procedure body. *SIGPLAN Not.*, 14(8):214–220, 1979.

[13] D.R. Chakrabarti and Shin-Ming Liu. Inline analysis: beyond selection heuristics. *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*, pages 12 pp.–, March 2006.

[14] Jan Hubicka. The gcc call graph module: A framework for inter-procedural optimization. In *Proceedings of the GCC Developers' Summit*, pages 65–78, 2004.

[15] Rainer Leupers and Peter Marwedel. Function inlining under code size constraints for embedded processors. In *In International Conference on Computer-Aided Design (ICCAD*, pages 253–256. IEEE Press, 1999.

[16] Robert Morgan. *Building an Optimizing Compiler*, chapter 9.2. Morgan Kaufmann Publishers, 2004.

[17] Steven S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, 1997.