ABSTRACT

Patil, Kaustubh S.: Compositional static cache analysis using module-level abstraction (Under the direction of Dr. Frank Mueller).

Static cache analysis is utilized for timing analysis to derive worst-case execution time of a program. Such analysis is constrained by the requirement of an inter-procedural analysis for the entire program. But the complexity of cycle-level simulations for entire programs currently restricts the feasibility of static cache analysis to small programs. Computationally complex inter-procedural analysis is needed to determine caching effects, which depend on knowledge of data and instruction references. Static cache simulation traditionally relies on absolute address information of instruction and data elements.

This thesis presents a framework to perform worst-case static cache analysis for direct-mapped instruction caches using a module-level and compositional approach, thus addressing the issue of complexity of inter-procedural analysis for an entire program. The module-level analysis parameterizes the data-flow information in terms of the starting offset of a module. The compositional analysis stage uses this parameterized data-flow information for each module. Thus, the emphasis here is on handling most of the complexity in the module-level analysis and performing as little analysis as possible at the compositional level. The experimental results show that the compositional analysis framework provides equally accurate predictions when compared with the simulation approach that uses complete inter-procedural analysis.

Compositional static cache analysis using module-level abstraction

by

Kaustubh S. Patil

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial satisfaction of the
requirements for the Degree of
Master of Science in Computer Science

Department of Computer Science

Raleigh

2003

Approved By:

____________________	____________________
Dr. Alexander Dean	Dr. Eric Rotenberg

____
Dr. Frank Mueller
Chair of Advisory Committee

Biography

Kaustubh S. Patil was born on 19th November 1979, in Mumbai, India. He received his Bachelor of Technology (Honors) degree in Computer Science and Engineering from the Indian Institute of Technology, Kharagpur, India in 2001. In fall of 2001, he came to the North Carolina State University to pursue graduate studies in computer science. He will receive the Master of Science degree in Computer Science from NCSU with the defense of this thesis.

Acknowledgements

This thesis was possible because of a lot of guidance and patience of my advisor Dr.Mueller. I would like to thank him immensely for all his support. I would also like to thank Dr.Eric Rotenberg and Dr.Alexander Dean for being on my advisory committee. I wish to thank Kiran Seth for providing the pathfile generators as well as helping in obtaining the experimental results. Thanks are due to Jaydeep Marathe who helped me in writing a thesis draft. Last but not the least I would like to thank my friends Raviraj Mahatme, Mangesh Dalvi, Harish Vishwanathan and Nipul Shah.

Contents

List of Figures **vi**

1 Introduction **1**

1.1 Static cache simulation 1

1.1.1 Control-flow graph generation 2

1.1.2 Instruction reference classifications 2

1.2 New compositional static cache analysis approach 3

1.2.1 Module-level analysis 3

1.2.2 Compositional analysis 3

1.3 Motivation 4

1.4 Organization of the document 4

2 Design of the Compositional Approach Framework **5**

2.1 Module-level Analysis 6

2.1.1 Module-level analysis ignoring calls 6

2.1.2 Module-level analysis considering calls 7

2.1.3 Module-level analysis with added outer loops and ignoring calls . . . 9

2.1.4 Module-level analysis with added outer loops and considering calls . 10

2.2 Compositional analysis 11

2.2.1 Need for module alignments 11

2.2.2 Initial predicted categorizations 12

2.2.3 Bottom-up processing 13

2.2.4 Adjustments to the categorizations as a callee 15

2.3 Summary 15

3 Examples Illustrating The Compositional Analysis Framework **19**

3.1 Timing analysis 19

3.2 Formulas for worst-case instruction categorizations 20

3.2.1 Definitions 20

3.2.2 Worst-Case Instruction Categorization 21

3.3 Example 1 of compositional analysis 22

3.3.1 Calculations in the traditional approach 23

3.3.2 Calculations in the new approach 24

3.4 Example 2 of compositional analysis 27
3.4.1 Calculations in the traditional approach 28
3.4.2 Calculations in the new approach 29
3.5 Example 3 of compositional analysis 31
3.5.1 Calculations in the traditional approach 31
3.5.2 Calculations in the new approach 32

4 Experimental validation **38**
4.1 Performance improvement 38
4.2 Accuracy of predicted categorizations 42
4.2.1 Comparison of worst-case execution cycles 42
4.2.2 Comparison of WCEC with Simplescalar simulator 44
4.3 Summary 45

5 Future Work **48**
5.1 Support for set-associative caches 48
5.2 Best-case categorizations 48
5.3 Multi-level modular analysis 49

6 Related Work **50**
6.1 Prior methods 50

7 Summary **53**

Bibliography **54**

List of Figures

1.1 Framework used in static cache and static timing analysis 1

2.1 Data-flow in the traditional integrated framework 5
2.2 Data-flow in the new framework 6
2.3 Module-level analysis ignoring calls 7
2.4 Situation simulated by module-level analysis ignoring calls 7
2.5 Module-level analysis considering calls 8
2.6 Situation simulated by module-level analysis considering calls 8
2.7 Module-level analysis with added outer-loops and ignoring calls 9
2.8 Situation simulated by module-level analysis with added outer-loops and ignoring calls 9
2.9 Module-level analysis with added outer loops and considering calls 10
2.10 Situation simulated by module-level analysis with added outer loops and considering calls 10
2.11 Example of a module aligned on a cache-line size boundary 11
2.12 Algorithm for bottom-up processing 16
2.13 Example of the effect of two different paths in a module on the exit lines . . 17
2.14 Algorithm for obtaining final categorizations 18

3.1 Compositional analysis example 1 35
3.2 Compositional analysis example 2 36
3.3 Compositional analysis example 3 37

4.1 Comparison of computation times for `adpcm` for cache line size=16 bytes . . 39
4.2 Comparison of computation times for `adpcm` for cache line size=32 bytes . . 40
4.3 Comparison of computation times for `mm` for cache line size=16 bytes 41
4.4 Comparison of computation times for `mm` for cache line size=32 bytes 42
4.5 Comparison of computation times for `mad` for cache line size=16 bytes . . . 43
4.6 Comparison of computation times for `mad` for cache line size=32 bytes . . . 44
4.7 Comparison of total compositional time with traditional time for `adpcm`, cache line size=16 bytes 45
4.8 Comparison of total compositional time with traditional time for `mm`, cache line size=16 bytes 46

4.9 Comparison of total compositional time with traditional time for `mad`, cache line size=16 bytes 47

Chapter 1

Introduction

This chapter introduces the method of static cache analysis [18], which is used in static timing analysis to accurately predict the worst-case execution time for a program. In this method of static cache simulation for instruction caches, an instruction reference is categorized as a hit or a miss. It uses control-flow partitioning and a function instance graph for categorizing every instruction reference. The framework used for the static cache analysis and the static timing analysis is shown in Figure 1.1.

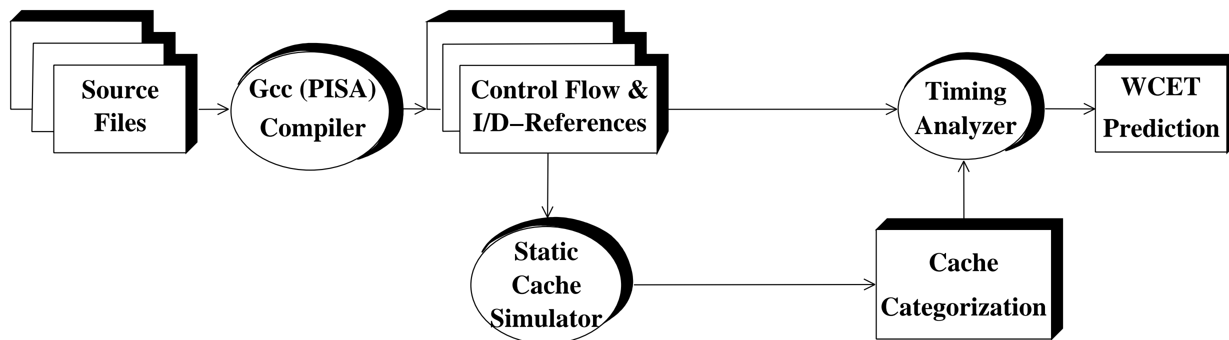


Figure 1.1: Framework used in static cache and static timing analysis

1.1 Static cache simulation

The excerpts of this section can be found in [4]. The method of static cache simulation is used to categorize every instruction reference before actual program execution for a specific cache configuration. It consists of three phases. In phase one, a control-flow

graph for the whole program is constructed. Then, this graph is analyzed to determine the program lines that can possibly be cached before entering each basic block of the program. In the last phase, this information is used to categorize each instruction reference.

1.1.1 Control-flow graph generation

Information about successors and predecessors of each basic block is generated using compiler back-end. Also, information about the function calls, if any, is collected for each basic block. Using this information, a control-flow graph is created for each function. Also, a function instance graph is generated for the whole program. An instance of a function refers to a unique invocation of that function. For example, if a function `foo` can be reached from the function `main` using two different control-flows, then `foo` has two instances in the function instance graph.

1.1.2 Instruction reference classifications

An instruction reference is categorized based on the abstract cache state of the basic block that the instruction lies in. For each loop nesting and function nesting level, one category is derived for each instruction.

Potentially cached line: A program line can potentially be cached before entering a basic block if there exists a sequence of control-flow such that the line is cached when the basic block is entered along that control-flow.

Abstract cache state: An abstract cache state of a basic block in a function instance is the subset of all program lines that can potentially be cached before the execution of that basic block.

An instruction reference is categorized into one of these four categories.

- **Always-Miss**: An instruction is categorized as an *always miss* if it cannot be guaranteed to be in cache for that reference. An *always miss* is predicted when the instruction is the first reference to a program line in the basic block and the program line is not in the abstract cache state of that basic block.

- **Always-Hit**: An instruction is categorized as an *always hit* if it is guaranteed to be in cache for that reference. An *always hit* is predicted when other instructions in the basic block have already accessed the same program line or the program line is in the

abstract cache state, and no other conflicting program line is in the abstract cache state.

- **First-Miss**: An instruction is categorized as a *first miss* if it cannot be guaranteed to be in cache the first time it is accessed when the loop is entered, but it is guaranteed to be in cache for all later iterations of that loop.
- **First-Hit**: An instruction is categorized as a *first hit* if it is guaranteed to be in cache the first time it is accessed when the loop is entered, but cannot be guaranteed for later iterations of that loop.

1.2 New compositional static cache analysis approach

This thesis proposes a method consisting of two stages to obtain categorizations for all instruction references in a program.

1.2.1 Module-level analysis

In module-level analysis, the control-flow information for only the module in question is used to obtain categorizations. The static cache simulation approach [18, 19] described in the previous section is used to perform module-level analysis. While doing module-level analysis, no absolute address information for instruction references is available. Four different types of analyses are performed on each module. These are described in the next chapter. Module-level analysis is comparable to the creation of an object file during compilation process, in which only relative address information is available for each module.

1.2.2 Compositional analysis

The second and the final stage of the new approach is the compositional analysis. In this stage, the absolute address information for each module is known prior to the analysis. Using the four different types of analyses performed during module-level analysis and a limited amount of inter-procedural analysis, the final categorizations for each instruction reference are arrived at. This stage is comparable to the linking of the object files during compilation when all the absolute address information for each object file is available.

1.3 Motivation

The static cache simulation approach described previously requires a time overhead comparable to that of data-flow analysis used in optimizing compilers, which can be quite high for large-sized programs involving complex control-flow patterns and nested structures. In the new approach, the burden of the time overhead is shifted to the module-level analysis, which needs to be done only once for each module, and it can be reused later for the complete analysis of any program that uses the module. Besides, the complexity of the second stage *i.e.* the compositional analysis, is lower than the traditional analysis of the entire program. Also the compositional approach predicts the categorizations without sacrificing any accuracy. The actual complexity estimations are explained in the next chapter.

In short, for the whole program, spacial locality cache effects are captured by cache simulation of module-intrinsic memory references during module-level analysis. During this first stage, the simulation of temporal locality is constrained to iterators within a module *i.e.* only a subset of temporal reuse can be calculated during the module-level analysis. The second stage *i.e.* the compositional analysis captures the effects of temporal locality on categorizations across calls to modules.

1.4 Organization of the document

This introductory chapter briefly describes the integrated static cache simulation approach and an outline of the new compositional approach. Next the design of the framework and the implementation issues involved are described in detail. This is followed by examples illustrating the compositional approach. Experimental results verifying the validity of this approach are mentioned in the following chapter. The document ends with a few remarks on the future work based on the new approach followed by related work and summary.

Chapter 2

Design of the Compositional Approach Framework

This chapter discusses the design of the new two-stage worst-case static cache analysis framework. Both the stages are described in detail. The new compositional approach works for a fixed cache configuration. *i.e.* for both, the module-level analysis and the compositional analysis, the cache configuration remains the same. Currently worst-case analysis for direct-mapped instruction caches is supported in the implementation.

The data-flow that takes place during the traditional integrated approach is depicted in Figure 2.1. This can be contrasted with the data-flow during the new approach which is shown in Figure 2.2.

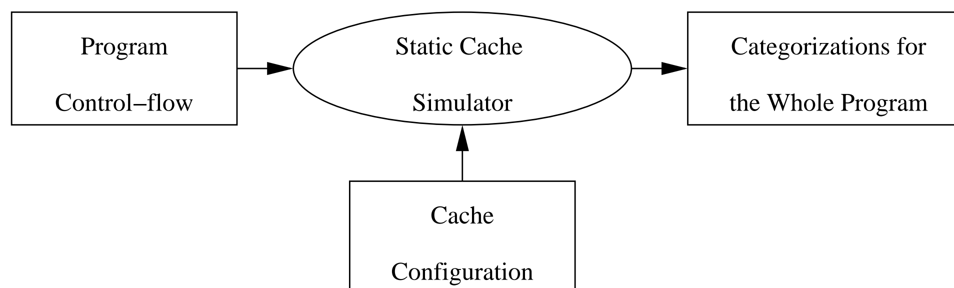


Figure 2.1: Data-flow in the traditional integrated framework

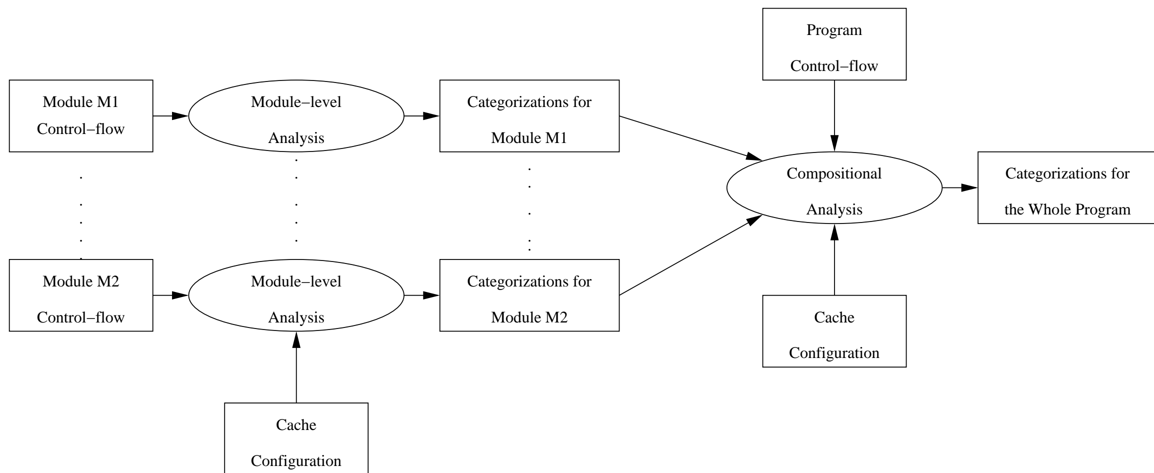


Figure 2.2: Data-flow in the new framework

2.1 Module-level Analysis

In the first stage, instructions are categorized only on the modular basis. During this stage, the module is considered to be laid out in its own address space starting with zero offset. The reason behind this requirement is explained in the next section. Also, at this level of analysis, no information is available about called modules, if any, except for the name of the modules. Four types of anlayses are performed on each module. These are described below. In all the examples of this section, the instruction cache is assumed to have four cache lines.

2.1.1 Module-level analysis ignoring calls

While performing abstract cache state (ACS) calculations, described in the previous chapter, for a module, it is assumed that if a predecessor of a basic block has a call to a function, the call will not affect the incoming abstract cache state for the basic block. This is equivalent to assuming that every line present in the abstract cache state of the predecessor block just before execution of the call will still remain in the cache even after the control returns to its successor block from the called module. *I.e.* the called module does not cache any line that is in conflict with the line in the ACS of the predecessor just before the execution of the call. Figure 2.3 gives an example of the ACS calculations for a module when calls are ignored.

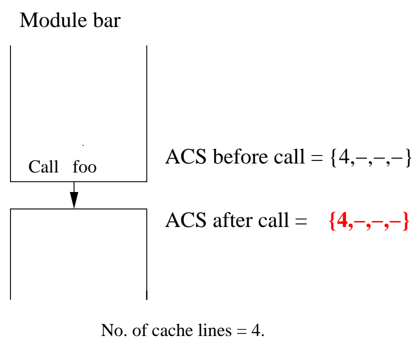


Figure 2.3: Module-level analysis ignoring calls

This analysis tries to capture the situation shown in Figure 2.4 that may occur during actual inter-procedural analysis.

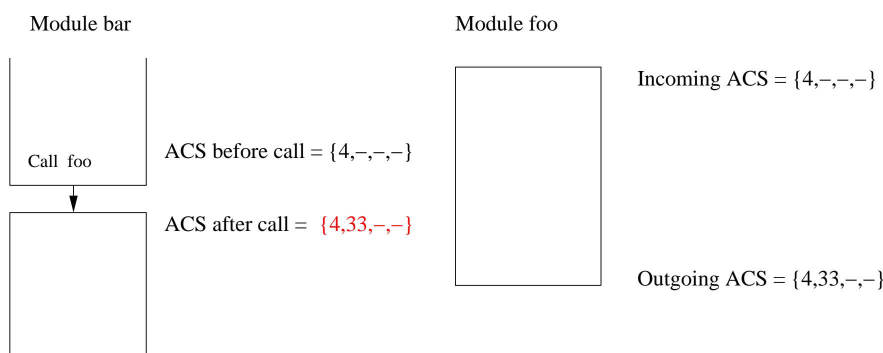


Figure 2.4: Situation simulated by module-level analysis ignoring calls

Every instruction reference for the module is categorized based on the ACS obtained using this assumption. Consider a program that uses this module. Consider a basic block B, such that its predecessor, P, has a call to a module. Consider a line l of the block P. If the called module does not cache in a line that maps to the same cache line as l, then for block B, the categorizations for an instruction of a line that maps to the same cache line as l will be exactly same as the one obtained for block B during the module-level analysis ignoring calls.

2.1.2 Module-level analysis considering calls

In this type of analysis, while calculating the ACS for each basic block of a module, it is assumed that if a block has a call to a module, then the called module caches lines that

map to every cache line of the cache. *i.e.* the called module evicts all the cache lines in the ACS of the calling block and caches in its own lines.

Since the absolute address information for the lines of the called module is not known during module-level analysis, we indicate a line of the called module that conflicts with a line in the ACS of the calling block as C (a must-conflict line). Figure 2.5 illustrates the underlying assumption for the module-level analysis considering effect of the calls on the caller.

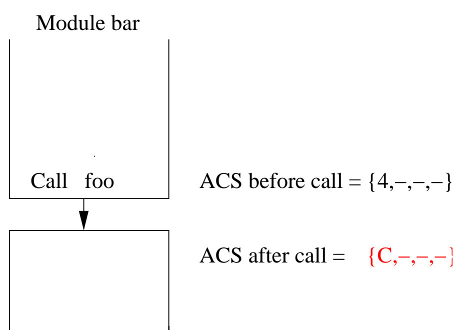


Figure 2.5: Module-level analysis considering calls

This analysis simulates the situation shown in Figure 2.6 that might occur during the actual inter-procedural analysis.

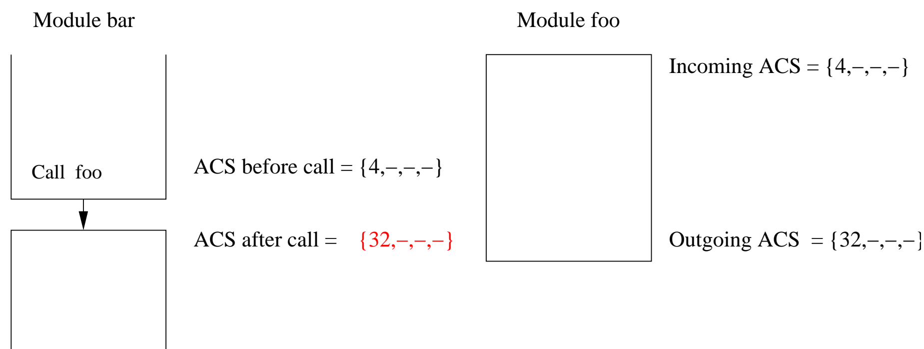


Figure 2.6: Situation simulated by module-level analysis considering calls

Consider a program that uses this module. Consider a basic block B such that its predecessor P has a call to a module. Consider a line l of the block P. If the called module caches a line that maps to the same cache line as l, then for block B, the categorizations for an instruction of a line that maps to the same cache line as l, will be exactly same as

the ones obtained for block B during the module-level analysis ignoring calls.

2.1.3 Module-level analysis with added outer loops and ignoring calls

Before performing this analysis on a module, we add a back-edge from each exit block to each entry block of this module. These added outer loops simulate the return of the control-flow to this module during a program execution. Also, during ACS calculations, effects of the calls are ignored as in the first subsection. Figure 2.7 illustrates modification of a control-flow graph after addition of back-edges. Figure 2.8 shows the situation that is simulated by this analysis.

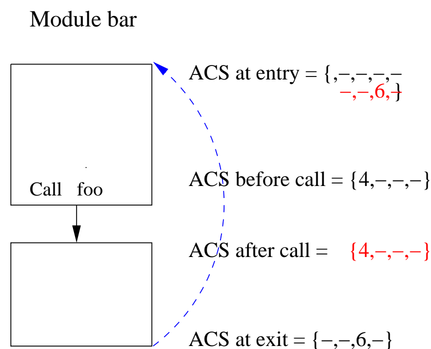


Figure 2.7: Module-level analysis with added outer-loops and ignoring calls

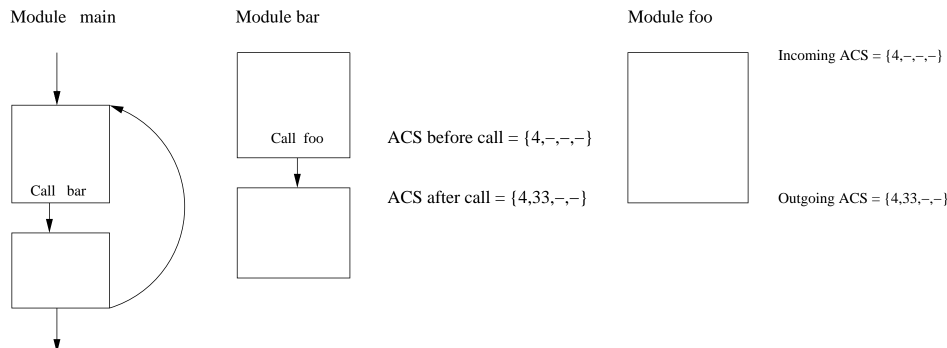


Figure 2.8: Situation simulated by module-level analysis with added outer-loops and ignoring calls

In this analysis, ACSs of the exit blocks of the module propagate along the added back-edges to the entry blocks of the module. Hence, the categorizations from this analysis are obtained with the same effect as if this module is being called from a loop in another

module, and the control returns to this module from the calling loop for each loop iteration except for the first iteration.

The calls are ignored during this analysis. This simulates a call within this module that does not have any effect on the incoming ACS of the successor of the calling basic block.

2.1.4 Module-level analysis with added outer loops and considering calls

As in the previous section, back-edges are added from exit blocks to entry blocks. But in this case, effects of the calls are considered. This simulates a module being called from a loop of another module, and the called module has a call that affects the incoming ACS of the successor of the calling block. Figure 2.9 illustrates the calculations during this analysis and Figure 2.10 shows the effects simulated by this analysis.

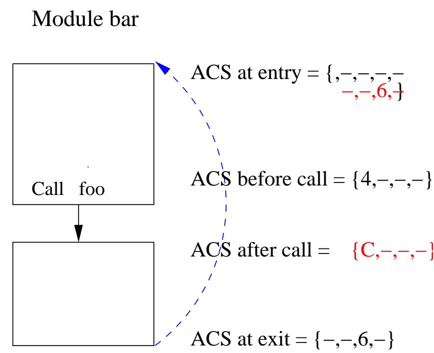


Figure 2.9: Module-level analysis with added outer loops and considering calls

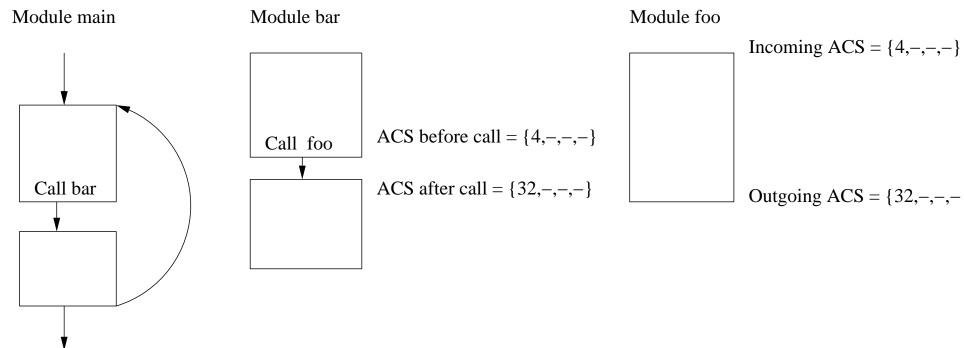


Figure 2.10: Situation simulated by module-level analysis with added outer loops and considering calls

In addition to these four different types of analyses, for each basic block the loop-

numbers to which the basic block belongs and for each loop, all the basic blocks that the loop contains are stored in files for that module.

2.2 Compositional analysis

Compositional analysis is performed on an entire program when the absolute address information for each module is available.

2.2.1 Need for module alignments

For the compositional analysis, it is assumed that a module is aligned on cache-line size boundary. *I.e.*, a module can only start on the beginning of a new line and not within a line. This is not a necessary but rather a simplifying assumption to reduce the overhead involved in the module-level analysis. This is explained below in detail. Figure 2.11 illustrates this assumption.

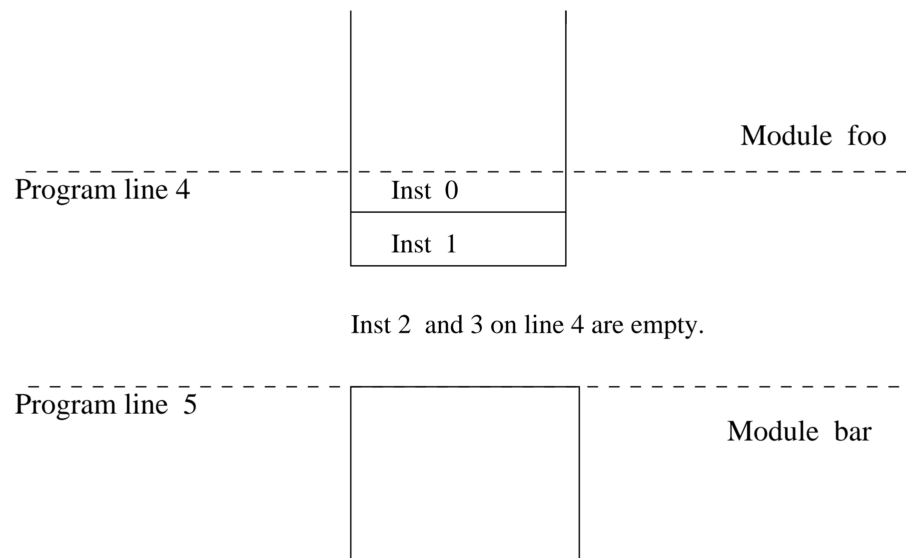


Figure 2.11: Example of a module aligned on a cache-line size boundary

If this requirement were not enforced, then a module could start in many positions w.r.t to a module above it in the process address space.The number of such possible combinations is *O(number of instructions in a cache line)*.

Suppose a module is laid out in the address space in such a way that it starts on the third instruction of a program line. Now, we cannot use the module-level analysis performed in the first stage for this module. The reason for this lies in the changed interaction of basic blocks within the module in terms of the cached lines passed during data-flow analysis. So for such a 'non-aligned' module, we need to use the module-level analysis, where we assume that the module starts on the third instruction of a program line. *i.e.* if we assume a four byte instruction, then the first instruction of this module will start from offset 8 during module-level analysis. If we do not enforce any restrictions on the starting offset of a module, we will need to perform the module-level analysis for each module *O(number of instructions in a cache line)* times. For a cache line size of 16 or 32 instructions, this imposes a significant overhead. Hence, the starting offset of a module is restricted to be a multiple of cache line size.

Besides, by imposing such an alignment for all modules, space wasted is at most $O(\frac{1}{2}nf \times cls)$ where *nf* is the number of functions and *cls* is the cache line size. Most current architectures that use instruction caches have a cache line size of 32, 64 or 128 bytes. Hence, the space wasted will not be a significant overhead on the memory used during program execution.

2.2.2 Initial predicted categorizations

For each instruction of each module in the program, it is assumed that the module-level categorizations ignoring calls are the initial categorizations. These are optimistic categorizations assuming that no temporal locality effects take place across module boundaries during control-flow transitions across calls. The compositional analysis is started with this assumption, and in the remaining processing, the necessary adjustments are applied to these categorizations.

Two kinds of adjustments are necessary for each line in each module.

Adjustments as a caller: When a basic block B has a predecessor P and P is a calling block, then for the block B, adjustments are needed using the *module-level analysis considering calls.*

Adjustments as a callee: When a module M is being called within a loop L of another module, then except the first iteration of L, the call to M within the remaining iterations might result in some lines of M still being present in the cache. Hence for those lines, we need to use the *module-level analysis with added outer loops.*

In the following subsection, the algorithm used is described and explained in detail.

2.2.3 Bottom-up processing

In the compositional analysis stage, a limited amount of analysis is done in order to obtain the inter-procedural relationship in terms of the cached lines passed across the module boundaries during calls and call returns. The modules are processed starting from the leaf module in a call graph and gradually traversing the graph toward the root *i.e.* the `main` function. The algorithm for a basic block used in the bottom-up processing is given in Figure 2.12.

As can be seen, the bottom-up processing is done for each module only once. The complexity of the above algorithm is $O(nf \times nlf \times nbl \times npb)$ where *nf* is the number of modules, *nlf* is the maximum number of loops in any one module, *nbl* is the maximum number of basic blocks in any one loop and *npb* is the maximum number of predecessors of a basic block. Thus, the complexity of this algorithm that captures the inter-procedural analysis information is kept at module-level. *I.e.*, for a module, there is no need to reanalyze its callees, thus, saving time overhead.

Lines belonging to each loop are calculated. Also each loop is assigned its own loopheader. The must_exit_lines for a module (mentioned in line 47 of the algorithm) store the lines that are guaranteed to be in cache when the control of execution returns from the module. The may_exit_lines for a module are the lines of that module that may be present in the cache on exit from the module, but their presence in cache cannot be guaranteed. For instance, if a line l of a module lies on one path P, but not on some other path P' of the module, and, following the control-flow along the path P, it is present in the outgoing lines of a specific exit block, then the line will **not** be present in the must_exit_lines of the module as it might not be cached if the control-flow goes along P'. It will be present in the may_exit_lines. This is further explained in the example given in the Figure 2.13.

As seen in the figure, when control is transitioned to the module `getbit`, line 43 may or may not be present as an exit line of `getbit`. If a block B in a module `des` calls

`getbit`, then for the successor S of block B, it is necessary to consider the fact that, when the control returns to S, line 43 of `getbit` might be in the cache, as we are considering the worst-case scenario. Hence, such a line is denoted by a notation **MC** *(a May-Conflict line)* in the exit lines of `getbit`.

Bottom-up processing gathers for each basic block, and for each loop that the block belongs to, the lines that are guaranteed to be in cache before control of execution enters that basic block (referred to as *must_incoming_lines* in Figure 2.12) and also the lines that cannot be guaranteed to be in cache before entering the basic block (referred to as *may_incoming_lines*). Line 15 of this algorithm shows the formula for calculating the *must_out_lines* for a basic block, if the block has a call. All the lines in the *must_out_lines* calculated on line 12 that are in conflict with any of the *may_exit_lines* of the callee are subtracted. Note that here the set *may_exit_lines* of the callee is used. This ensures that the set *must_out_lines* calculated is safe and accurate. Then all the lines in *must_exit_lines* of the callee are added, because these lines are guaranteed to be in cache when the control returns from the callee. Formula on line 22 similarly calculates the *may_out_lines*.

As can be seen from line 31 of this algorithm, this information is used to adjust the categorizations for a module when it acts as a caller. When a block has a call to a module, its successor block might have some lines of the module as its incoming lines. In that case, the categorizations for those lines of the successor that conflict with incoming lines of the module need to be adjusted. This is done using the *module-level analysis considering calls*. For adjusting the categorizations with added outer loops, a similar analysis (not described in the figure) is performed for each function and each loop, but with an added back-edge from each exit block to the entry block of the function. Thus, the adjustments to the initial categorizations required for a module when a module acts as a caller are applied during the bottom-up processing. It is important to understand here that these adjustments will change the categorizations from an overly optimistic to a realistic and safe (conservative) worst-case value. The module may also act as a callee. In that case, some lines of the module may still be in cache when the flow of control returns to the module on every loop iteration except the first one. This will cause subsequent hits for such lines. These adjustments are applied during the second pass when each instruction is categorized w.r.t each loop it lies in and w.r.t each instance of the module. This is achieved by using information of the incoming lines for each calling block of the module and also the *adjusted module-level analysis with added outer loops*. This is explained in the next subsection.

2.2.4 Adjustments to the categorizations as a callee

During this second and final pass of the compositional analysis, categorizations of each instruction might be adjusted depending upon whether the module is called by some other module or not. The algorithm for this pass is given in Figure 2.14. This algorithm is used for each line l in the program.

In the *do while();* loop of this algorithm we try to find out that, when the control returns to line l of the module from the call within loop L, if line l is still in cache. If it is, then it is going to be a hit, and we should use the categorizations from the simulated analysis with added outer loops.

2.3 Summary

The new compositional static cache categorizations approach uses two stages. In the first stage, analysis is done on module-levels without knowledge of the absolute addresses of the module and those of the called modules. In the second stage a two-pass approach is used. In the first pass, information is gathered in a bottom-up manner about incoming lines for each basic block and the exit lines of each module. Adjustments to the initial categorizations when the module acts as a caller are applied in this pass. In the second pass, each instruction is categorized for each loop level and each instance of the module. Also, adjustments are applied when the module acts as a callee. The complexity is kept at module levels for the compositional approach.

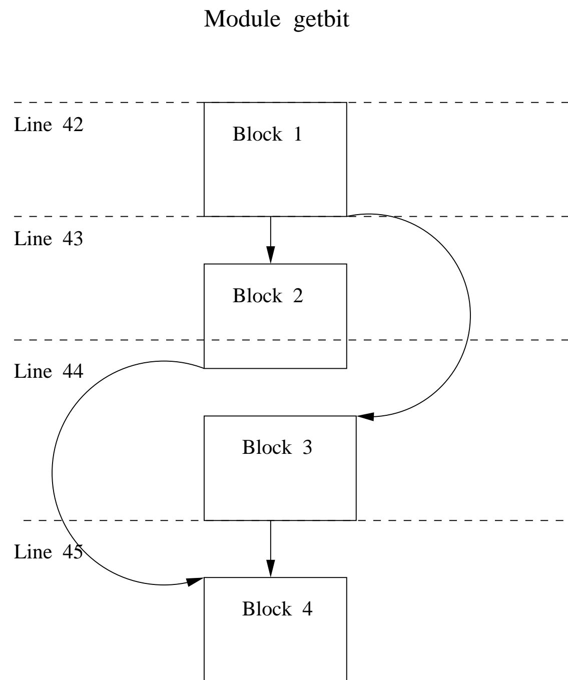
```
for (each function F)
   if (isleaf_function || all children processed) { // Ensures bottom-up order
      for (each loop L in F) {
         /* Calculations for incoming lines of each block of this loop */
         for (each block B in the loop L){
            inter_out = full set;
            union_out = empty set;
            for (each predecessor P of B in loop L) {
               if ( B!=L → loopheader)
                  if (backedge(P-to-B)) // P-to-B form a loop inside L. Disregard this predecessor
                     continue;
12:            must_out_lines(P)=must_incoming_lines(P)-conf_prog_lines(P)+prog_lines(P);
               if (iscaller(P)){
                  function C = P→callee;
15:               must_out_lines(P)=must_out_lines(P)-conf_prog_lines(C→may_exit_lines)+prog_lines(C→must_exit_lines);
               }
               inter_out=intersection(must_out_lines(P),inter_out);
               // Note that for the very first pred, inter_out above is calculated as union
               may_out_lines(P)=may_incoming_lines(P)-conf_prog_lines(P)+prog_lines(P);
               if (iscaller(P)){
                  function C = P→callee;
22:               may_out_lines(P)=may_out_lines(P)-conf_prog_lines(C→must_exit_lines)+prog_lines(C→may_exit_lines);
               }
               union_out=union(may_out_lines(P),union_out);
            }
            copy(must_incoming_lines(B),inter_out);
            copy(may_incoming_lines(B),union_out);
         } // Processing for block B complete

         /* Adjustments applied as a caller */
31:      for (each block B in L){
            for (each line l in B)
               if (l conflicts with some line in incoming_line(B))
                  for (each instruction in l)
                     normal_category = module-level cat considering call;
         }
         /* Calculations of loopheader and lines within this loop */
         find loopheader of loop L;
         lines(loop L) = cache all the lines of each block of this loop;
         if (loop L has calls)
            for (each callee)
               lines(loop L) = lines(loop L)+lines(callee);

      } // Processing for loop L complete

      // calculate the must and may exit_lines for this function.
47:   F→must_exit_lines = full set;
      F→may_exit_lines = empty set;
      for (each exit block E){
50:      F→must_exit_lines =∩ must_incoming_lines(E)-conf_prog_lines(E)+prog_lines(E);
         F→may_out_lines =∪ may_incoming_lines(E)-conf_prog_lines(E)+prog_lines(E);
      }
   }
```

Figure 2.12: Algorithm for bottom-up processing



Line 43 is cached along one path and not cached along other.

Figure 2.13: Example of the effect of two different paths in a module on the exit lines

```
// This algorithm is for a particular instance of the module.

for (each loop L with header H the line l lies in) {
  l_in_conflict=FALSE;
  calling block C = basic block that contains l;
  do {
    if (l is in conflict with a line l' in must_incoming_lines(C) || may_incoming_lines(C)) {
      l_in_conflict = TRUE;
      break;
    }
    C = the block that calls the instance of the module containing current block C;
  } while(C → currentmodule != H → currentmodule);

  for (each instruction i of l)
    if (!l_in_conflict)
      i.finalcat = adjusted categorizations obtained from module-level analysis with added outer-loops;
    else
      i.finalcat = adjusted categorizations obtained from module-level analysis without added outer-loops;
}
```

Figure 2.14: Algorithm for obtaining final categorizations

Chapter 3

Examples Illustrating The Compositional Analysis Framework

This chapter focuses on the description of three examples illustrating the various stages of the compositional approach introduced in the previous chapter. Source code for each example is given followed by the control-flow graph and the calculations for finding incoming lines and exit lines for each module. We start with explaining how the categorizations derived from this framework are used to measure timing effects of instruction caches during timing analysis and follow up with the examples. By these examples, we explain the basic terms and notations used in this chapter.

3.1 Timing analysis

Excerpts of this section can be found in [6, 19]. The timing analyzer constructs a timing tree by traversing all paths within a loop level, and propagating this timing information bottom-up within the tree. The timing tree represents the calling structure and the loop structure of the program. Functions are distinguished by their calling paths into function instances. Each function instance is regarded as a loop level with one iteration and is represented as a node in the timing tree. Regular loops are represented as child nodes of their surrounding function instance or as child nodes of another loop that they are nested in.

The timing analyzer determines the worst-case execution time (WCET) in a bottom-up traversal of the tree. For any node, all possible paths within the current loop level are

analyzed. When a child node, which represents a nested loop or invoked function, is encountered along a path, its WCET is already calculated and can simply be added to the WCET of the current path after multiplying it with the number of iterations of the current nesting level. Adjustments are necessary for transitions from (1) first-misses to first-misses and (2) always-misses to first-hits between loop levels since the caching behavior of the instructions within these transitions is not the same for each invocation of the inner loop. Further comprehensive details of the timing analysis can be found in [6, 19].

3.2 Formulas for worst-case instruction categorizations

The definitions and formulas in this section are directly used from [19].

3.2.1 Definitions

- **Linear Cache State (LCS)** The linear cache state of a program line l within a basic block and a function instance is the set of program lines that can potentially be cached in the forward control-flow graph prior to the execution of l within the basic block and the function instance.

  Informally, the LCS represents the hypothetical cache state in the absence of loops. This data-flow information represents the lines that may be cached between the entry of a loop to the current basic block on the first iteration. It will be used to determine whether a program line may be cached due to loops or due to the sequential control flow.

- **Dominator Cache State (DCS)** The dominator cache state of a program line l within a basic block and a function instance is the set of program lines that are known be cached prior to the execution of l within the basic block and the function instance.

  The DCS can be used to determine if a program line must be cached due to an earlier reference, regardless of the sequence of execution paths that were taken.

- **Post-dominator Set (PDS)** The post-dominator set of a program line l within a basic block and a function instance is the self-reflexive transitive closure of post-dominating program lines.

  This information is commonly used wrt. basic blocks in optimizing compilers.

3.2.2 Worst-Case Instruction Categorization

This subsection describes the formulas used in obtaining worst-case categorizations. These formulas are based on the definitions given in the previous subsection.

- Let i_k be an instruction within a path, a loop λ, and a function instance.
- Let n be the degree of associativity of the cache.
- Let $l = i_0..i_{m-1}$ be the program line containing i_k and let i_{first} be the first instruction of l within the path.
- Let s_j be the j-th component of the ACS (n-tuple) for l within the path and let $s = \bigcup_{1 \leq j \leq n} s_j$.
- Let l map into cache line c, denoted by $l \rightarrow c$.
- Let u be the set of program lines in loop λ.
- Let $child(\lambda)$ be the child loop (inner-next loop within nesting) of λ for this path and function instance, if such a child loop exists.
- Let $headers(\lambda)$ be the set of header paths and $preheaders(\lambda)$ be the set of preheader paths of loop λ, respectively.[1]
- Let $s(p)$ be the abstract output cache state of path p.
- Let $linear_j$ be the j-th component of the LCS (n-tuple) for l within the path and let $linear = \bigcup_{1 \leq j \leq n} linear_j$.
- Let dom be the set of dominating program lines (DCS) of path p.
- Let $postdom(p)$ be the set of self-reflexive post-dominating program lines of path p.

Then, $category(i_k, \lambda) =$

$$
\begin{cases}
\text{always-hit} & \text{if } k \neq \mathit{first} \vee (l \in \mathit{dom} \wedge \\
& \quad [\underset{1 \leq j \leq n}{\exists} \; l \in s_j \wedge (\underset{m \to c, m \neq l}{\Sigma} |m \in s_j| = 0 \vee \underset{m \to c, m \neq l}{\Sigma} |m \in s| < n)]) \\
\text{first-hit} & \text{if } \mathit{category}(i_k, \mathit{child}(\lambda)) = \text{first-hit} \vee k = \mathit{first} \wedge l \in s \wedge l \in \mathit{dom} \wedge \\
& \quad \underset{p \in \mathit{headers}(\lambda)}{\forall} \; l \in \mathit{postdom}(p) \wedge \underset{m \to c, m \neq l}{\Sigma} |m \in (s \cap u)| \geq n \wedge \\
& \quad \underset{\substack{m \to c, m \neq l \\ p \in \mathit{preheaders}(\lambda)}}{\Sigma} |m \in ((s(p) \cup \mathit{linear}) \cap u)| < n \\
\text{first-miss} & \text{if } \mathit{category}(i_k, \mathit{child}(\lambda)) = \text{first-miss} \wedge k = \mathit{first} \wedge l \in s \wedge \\
& \quad \underset{m \to c, m \neq l}{\Sigma} |m \in s| \geq n \wedge \underset{m \to c, m \neq l}{\Sigma} |m \in (s \cap u)| < n \\
\text{always-miss} & \text{otherwise}
\end{cases}
$$

3.3 Example 1 of compositional analysis

```
int max(int,int);
int a[10];
void main(){
        int i;
        for(i=0;i<10;i++)
                a[i]=max(i,i);
}
int max(int index,int i){
        int high;
        if(a[index]>i)
                high=a[index];
        else
                high=i;
        return high;
}
```

This program sets the array elements to the maximum of the indices or the initial value of the elements themselves. The control-flow graph for this program is shown in Figure 3.1.

3.3.1 Calculations in the traditional approach

The traditional calculations of abstract cache states are given below. In these calculations, line "**I**" indicates an invalid line. When the analysis begins, it is assumed that all the lines of the ACS are invalid lines. Also if an ACS is represented as $[a\ b\ c\ d]$, this means program line a is mapped to cache line 0, line b is mapped to cache line 1 and so on. It should be noted that calculations for linear cache states (LCS), dominator cache states (DCS) and post-dominator sets (PDS) defined earlier are also performed in a similar way, but these calculations are not shown in these examples.

In all the examples the cache is assumed to contain four lines.

```
Pass 1:
in(1) = [I I I I]        out(1) = [0 1 I I]
in(2) = [0 1 I I]        out(2) = [0 1 2 I]
in(5) = [0 1 2 I]        out(5) = [8 1 6 7]
in(6) = [8 1 6 7]        out(6) = [8 9 10 7]
in(7) = [8 1 6 7]        out(7) = [8 1 10 11]
in(8) = [8 9 10 7        out(8) = [8 9 10 11
           1    11]                   1      ]
in(3) = [8 9 10 11       out(3) = [4 5 2 3]
           1      ]
in(4) = [0 1 I I         out(4) = [0   I I
         4 5 2 3]                  4 5 2 3]
Pass 2:
in(1) = [I I I I]       out(1) = [0 1 I I]
in(2) = [0 1 I I        out(2) = [0 1 2 I
         4 5 2 3]                    4   3]
in(5) = [0 1 2 I        out(5) = [8 1 6 7]
         4     3]
in(6) = [8 1 6 7]       out(6) = [8 9 10 7]
```

```
in(7) = [8 1 6 7]       out(7) = [8 1 10 11]
in(8) = [8 9 10 7       out(8) = [8 9 10 11
           1    11]                 1      ]
in(3) = [8 9 10 11      out(3) = [4 5 2 3]
           1      ]
in(4) = [0 1 I I         out(4) = [0   I I
          4 5 2 3]                 4 5 2 3]
```

The first column of categorizations in Figure 3.1 is based on these calculations. Our ultimate aim is to achieve the exact same categorizations using the new method.

3.3.2 Calculations in the new approach

Step 1: Calculations ignoring all the calls

Here we assume that the calls make no change in the cache states of the basic blocks in the caller. For `main()` and `max()`, we show the calculations below.

```
main():
Pass 1:
in(1) = [I I I I]       out(1) = [0 1 I I]
in(2) = [0 1 I I]       out(2) = [0 1 2 I]
in(3) = [0 1 2 I]       out(3) = [4 5 2 3]
in(4) = [0 1 I I        out(4) = [0   I I
          4 5 2 3]                4 5 2 3]
Pass 2:
in(1) = [I I I I]       out(1) = [0 1 I I]
in(2) = [0 1 I I        out(2) = [0 1   I
          4 5 2 3]                4   2 3]
in(3) = [0 1   I        out(3) = [4 5 2 3]
          4   2 3]
in(4) = [0 1 I I        out(4) = [0   I I
          4 5 2 3]                4 5 2 3]

max():
```

```
in(5) = [I I I I]        out(5) = [8 I 6 7]
in(6) = [8 I 6 7]        out(6) = [8 9 10 7]
in(7) = [8 I 6 7]        out(7) = [8 I 10 11]
in(8) = [8 9 10 7        out(8) = [8 9 10 11
           I    11]                  I      ]
```

We begin by assuming that the categorizations derived using these calculations (shown with "W/o calls" heading in Figure 3.1) are the initial categorizations and proceed by adjusting these to reflect the correct values.

Step 2: Adjustments for the caller

For the caller (`main` in this example) the possible differences in calculations may arise when the control returns to it after the callee is executed. Some cache lines in the cache state before the call might be evicted by the callee.

To handle this issue, first we need the ACS calculations for the caller with the assumption that after each call, the ACS is invalidated.

eg. for main():

```
Pass 1:
in(1) = [I I I I]         out(1) = [0 1 I I]
in(2) = [0 1 I I]         out(2) = [0 1 2 I]
// Return after the call
// Note the different connotations of I and C.
in(3) = [C C C C]         out(3) = [4 5 2 3]
in(4) = [0 1 I I          out(4) = [0   I I
         4 5 2 3]                   4 5 2 3]
Pass 2:
in(1) = [I I I I]         out(1) = [0 1 I I]
in(2) = [0 1 I I          out(2) = [0 1   I
         4 5 2 3]                   4   2 3]
// Return after the call
in(3) = [C C C C]         out(3) = [4 5 2 3]
in(4) = [0 1 I I          out(4) = [0   I I
         4 5 2 3]                   4 5 2 3]
```

Now, we focus on the exit lines of the callee (`max`). It should be noted here that though the line 9 is cached along one path it is not cached along the other. Hence, it does not appear in the exit lines but instead a line denoted by MC (as described in the previous chapter) is present in the exit lines. In this example, these are

```
out(8) = [8 MC 10 11]
```

This shows that the lines in the caller (`main`), which conflict with lines 8, MC, 10 and 11 of the callee, will certainly be evicted by the callee. We consider those lines in `main` after the call site, which are in conflict with lines 8, MC, 10 and 11 of the callee, and for these lines we use the categorizations based on the calculations above as the final categorizations.

In general, we consider the conflicting lines between two call sites and adjust the categorizations. This approach completely adjusts the categorizations of the function as a caller.

Step 3: Adjustments for the callee

We need to calculate the ACS with the assumption that there is a back-edge from each exit block to the entry block of the callee function. In this example, these are not shown, but it is easy to see that all the instructions will be categorized as hits.

The call to max() in block 2 lies inside the loop of blocks 2 and 3. Now, we perform the ACS calculations only for this loop assuming an invalid ACS in the beginning of the header block (in this case, block 2) and ignoring the call. This yields

```
Pass 1:
in(2) = [I I I I]        out(2) = [I 1 2 I]
in(3) = [I 1 2 I]        out(3) = [4 5 2 3]
Pass 2:
in(2) = [4 5 2 3]        out(2) = [4 1 2 3]
in(3) = [4 1 2 3]        out(3) = [4 5 2 3]
```

This shows that just before the second call to `max`, no line of `max` will be present in the ACS, as there is no [I] in the output ACS of block 3. Hence, the back-edge analysis need not be used for the callee and the analysis assuming invalid ACS in the beginning for the callee should suffice. On the other hand, if there was an [I] in the output ACS of block 3, we must use the back-edge analysis for that particular line in the callee, which conflicts with this [I]. This covers the adjustments required for the callee function.

3.4 Example 2 of compositional analysis

In this example we have a call graph of height 2. The code is as follows:

```
#include<stdio.h>
int a[10];
int rand(int);
int pseudo_rand(int);

int main(){
        int i;
        for(i=0;i<10;i++)
              a[i]=rand(i);
        return 0;
}

int rand(int i){
        int val;
        if((val=pseudo_rand(i)) > 50)
                return val;
        else
                return 100-val;
}

int pseudo_rand(int i){
        return 7*i+7;
}
```

The control-flow graph is depicted in Figure 3.2. The figure also lists the categorizations based on calculations with different assumptions.

Here most of the explanation for deriving the final categorizations remains the same, so we focus on the cache state calculations to check the validity of the approach.

3.4.1 Calculations in the traditional approach

The traditional ACS calculations are given below:

```
Pass 1:
in(1) = [I I I I]        out(1) = [0 1 I I]
in(2) = [0 1 I I]        out(2) = [0 1 I I]
in(5) = [0 1 I I]        out(5) = [0 1 6 I]
in(9) = [0 1 6 I]        out(9) = [12 1 6 11]
in(6) = [12 1 6 11]      out(6) = [8 1 6 7]
in(7) = [8 1 6 7]        out(7) = [8 1 6 7]
in(8) = [8 1 6 7]        out(8) = [8 9 10 7]
in(3) = [8 9 10 7]       out(3) = [4 5 2 3]
in(4) = [0 1 I I         out(4) = [0   I I
          4 5 2 3]                  4 5 2 3]


Pass 2:
in(1) = [I I I I]        out(1) = [0 1 I I]
in(2) = [0 1 I I         out(2) = [0 1 I I
          4 5 2 3]                  4   2 3]
in(5) = [0 1 I I         out(5) = [0 1 6 I
          4   2 3]                  4     3]
in(9) = [0 1 6 I         out(9) = [12 1 6 11]
          4     3]
in(6) = [12 1 6 11]      out(6) = [8 1 6 7]
in(7) = [8 1 6 7]        out(7) = [8 1 6 7]
```

```
in(8) = [8 1 6 7]        out(8) = [8 9 10 7]
in(3) = [8 9 10 7]       out(3) = [4 5 2 3]
in(4) = [0 1 I I         out(4) = [0   I I
         4 5 2 3]                  4 5 2 3]
```

For the main function, the ACS calculations assuming 1) no calls and 2) with calls and ACS invalidated on return are as follows:

3.4.2 Calculations in the new approach

Step 1: Calculations ignoring all the calls

```
Pass 1:
in(1) = [I I I I]        out(1) = [0 1 I I]
in(2) = [0 1 I I]        out(2) = [0 1 I I]
in(3) = [0 1 I I]        out(3) = [4 5 2 3]
in(4) = [0 1 I I         out(4) = [0   I I
         4 5 2 3]                  4 5 2 3]


Pass 2:
in(1) = [I I I I]        out(1) = [0 1 I I]
in(2) = [0 1 I I         out(2) = [0 1 I I
         4 5 2 3]                  4   2 3]
in(3) = [0 1 I I         out(3) = [4 5 2 3]
         4   2 3]
in(4) = [0 1 I I         out(4) = [0   I I
         4 5 2 3]                  4 5 2 3]
```

Step 2: Adjustments for the caller

```
Calculations considering calls and invalid ACS on return


Pass 1:
in(1) = [I I I I]        out(1) = [0 1 I I]
```

```
in(2) = [0 1 I I]        out(2) = [0 1 I I]
in(3) = [C C C C]        out(3) = [4 5 2 3]
in(4) = [0 1 I I         out(4) = [0   I I
          4 5 2 3]                  4 5 2 3]


Pass 2:
in(1) = [I I I I]        out(1) = [0 1 I I]
in(2) = [0 1 I I         out(2) = [0 1 I I
          4 5 2 3]                  4   2 3]
in(3) = [C C C C]        out(3) = [4 5 2 3]
in(4) = [0 1 I I         out(4) = [0   I I
          4 5 2 3]                  4 5 2 3]
```

For the function rand( ), the calculations are as below:

```
Calculations assuming no calls

in(5) = [I I I I]        out(5) = [I I 6 I]
in(6) = [I I 6 I]        out(6) = [8 I 6 7]
in(7) = [8 I 6 7]        out(7) = [8 I 6 7]
in(8) = [8 I 6 7]        out(8) = [8 9 10 7]

Calculations considering calls and invalid ACS on return

in(5) = [I I I I]        out(5) = [I I 6 I]
in(6) = [C C C C]        out(6) = [8 C C 7]
in(7) = [8 C C 7]        out(7) = [8 C C 7]
in(8) = [8 C C 7]        out(8) = [8 9 10 7]
```

Finally for pseudorandom( ):

```
in(9) = [I I I I]        out(9) = [12 I I 11]
```

The categorizations based on these calculations and various phases of the analysis are shown along the control-flow graph for the example.

3.5 Example 3 of compositional analysis

This small example has a conditional call. The code is as follows:

```
#include<stdio.h>

int a[] = {1,2,3,4,5,6,7,8,9,10};

int value (int i)
{
  return a[i];
}

int main()
{
  int sum = 0, i;
  for(i=0; i<10; i++)
    if(i)
      sum += value(i);

  return sum;
}
```

The control-flow graph is depicted in Figure 3.3. The figure also lists the categorizations based on different calculations.

As in the previous section, we focus on the cache state calculations to check the validity of the approach.

3.5.1 Calculations in the traditional approach

The traditional ACS calculations are given below:

```
Pass 1:
in(2) = [I I I I]        out(2) = [I I 2 3]
in(3) = [I I 2 3]        out(3) = [4 I 2 3]
```

```
in(6) = [4 I 2 3]        out(6) = [4 I 6 7]
in(1) = [4 I 6 7]        out(1) = [0 1 6 7]
in(7) = [0 1 6 7]        out(7) = [0 1 6 7]

in(4) = [4 I 2 3         out(4) = [4 I 2 3
          0 1 6 7]                   1 6 7]
in(5) = [4 I 2 3         out(5) = [4 5 6 3
            1 6 7]                       7]
Pass 2:
in(2) = [I I I I]        out(2) = [I I 2 3]
in(3) = [I I 2 3         out(3) = [4 I 2 3
          4 1 6 7]                   1 6  ]
in(6) = [4 I 2 3         out(6) = [4 I 6 7
            1 6  ]                   1    ]
in(1) = [4 I 6 7         out(1) = [0 1 6 7]
            1    ]
in(7) = [0 1 6 7]        out(7) = [0 1 6 7]
in(4) = [4 I 2 3         out(4) = [4 I 2 3
          0 1 6 7]                   1 6 7]
in(5) = [4 I 2 3         out(5) = [4 5 6 3
            1 6 7]                       7]
```

For the `main` function, the ACS calculations assuming 1) no calls and 2) with calls and ACS invalidated on return are as follows:

3.5.2 Calculations in the new approach

Step 1: Calculations ignoring all the calls

```
Pass 1:
in(2) = [I I I I]        out(2) = [I I 2 3]
in(3) = [I I 2 3]        out(3) = [4 I 2 3]
in(6) = [4 I 2 3]        out(6) = [4 I 6 7]
in(7) = [4 I 6 7]        out(7) = [4 I 6 7]
in(4) = [4 I 2 3         out(4) = [4 I 2 3
```

```
             6 7]                       6 7]
in(5) = [4 I 2 3         out(5) = [4 5 6 3
             6 7]                         7]
```

```
Pass 2:
in(2) = [I I I I]        out(2) = [I I 2 3]
in(3) = [I I 2 3         out(3) = [4 I 2 3
         4   6 7]                   6  ]
in(6) = [4 I 2 3         out(6) = [4 I 6 7]
             6  ]
in(7) = [4 I 6 7]        out(7) = [4 I 6 7]
in(4) = [4 I 2 3         out(4) = [4 I 2 3
             6 7]                   6 7]
in(5) = [4 I 2 3         out(5) = [4 5 6 3
             6 7]                     7]
```

Step 2: Adjustments for the caller

```
Calculations considering calls and invalid ACS on return
```

```
Pass 1:
in(2) = [I I I I]        out(2) = [I I 2 3]
in(3) = [I I 2 3]        out(3) = [4 I 2 3]
in(6) = [4 I 2 3]        out(6) = [C C C C]
in(7) = [C C C C]        out(7) = [C C C 7]
in(4) = [4 I 2 3         out(4) = [4 I 2 3
         C C C 7]                   C 7]
in(5) = [4 I 2 3         out(5) = [4 5 6 3
             C 7]                     7]
Pass 2:
in(2) = [I I I I]        out(2) = [I I 2 3]
in(3) = [I I 2 3         out(3) = [4 I 2 3
         4   C 7]                   C  ]
```

```
in(6) = [4 I 2 3          out(6) = [C C C C]
             C  ]
in(7) = [C C C C]         out(7) = [C C C 7]


in(4) = [4 I 2 3          out(4) = [4 I 2 3
         C C C 7]                       C 7]
in(5) = [4 I 2 3          out(5) = [4 5 6 3
             C 7]                         7]
```

For the function value( ), the calculations are as below. Calculations assuming no calls and calculations considering calls and invalid ACS on return are the same.

```
in(1) = [I I I I]         out(1) = [0 1 I I]
```

The categorizations based on these calculations and various phases of the analysis are shown along the control-flow graph in Figure 3.3.

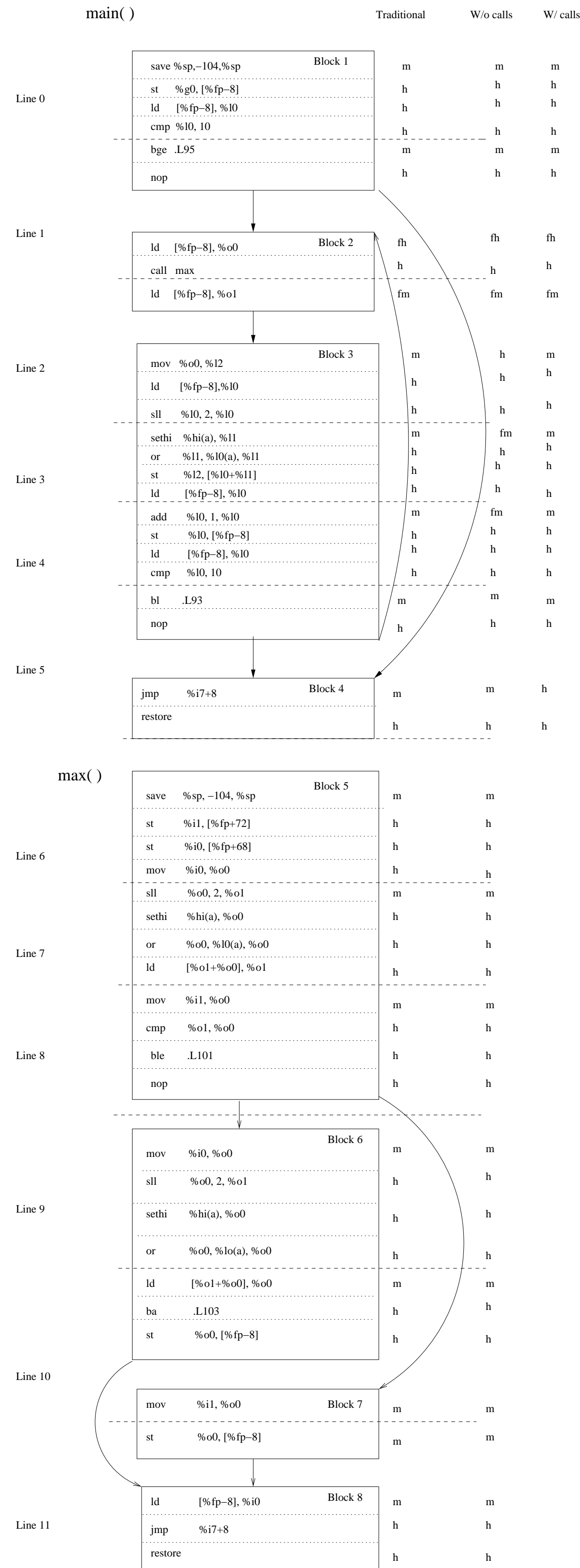


Figure 3.1: Compositional analysis example 1

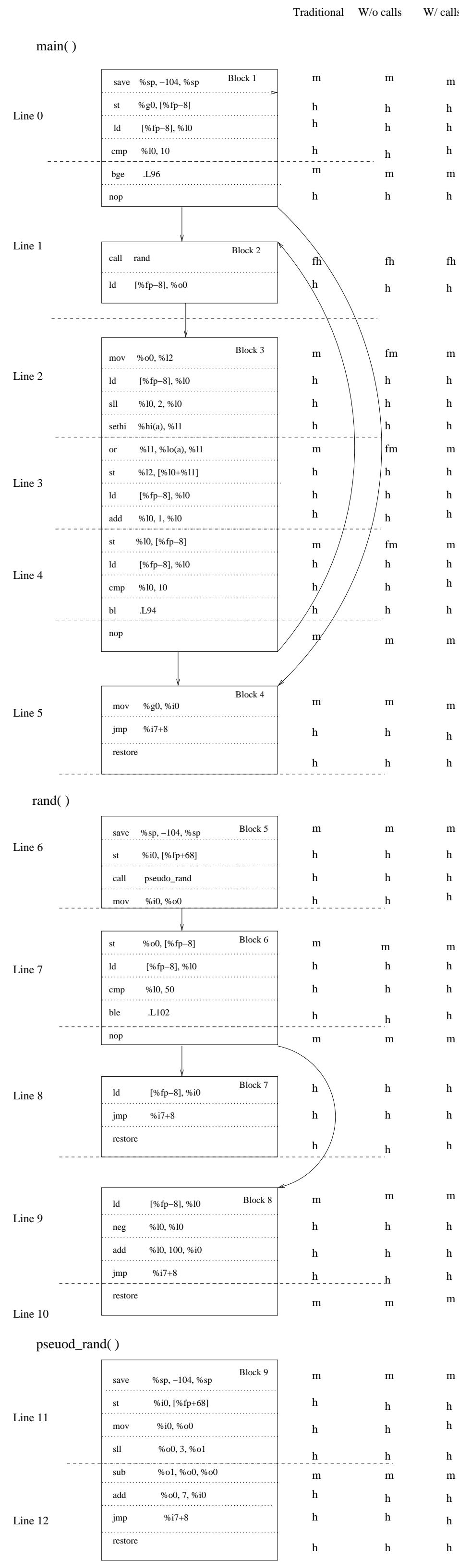


Figure 3.2: Compositional analysis example 2

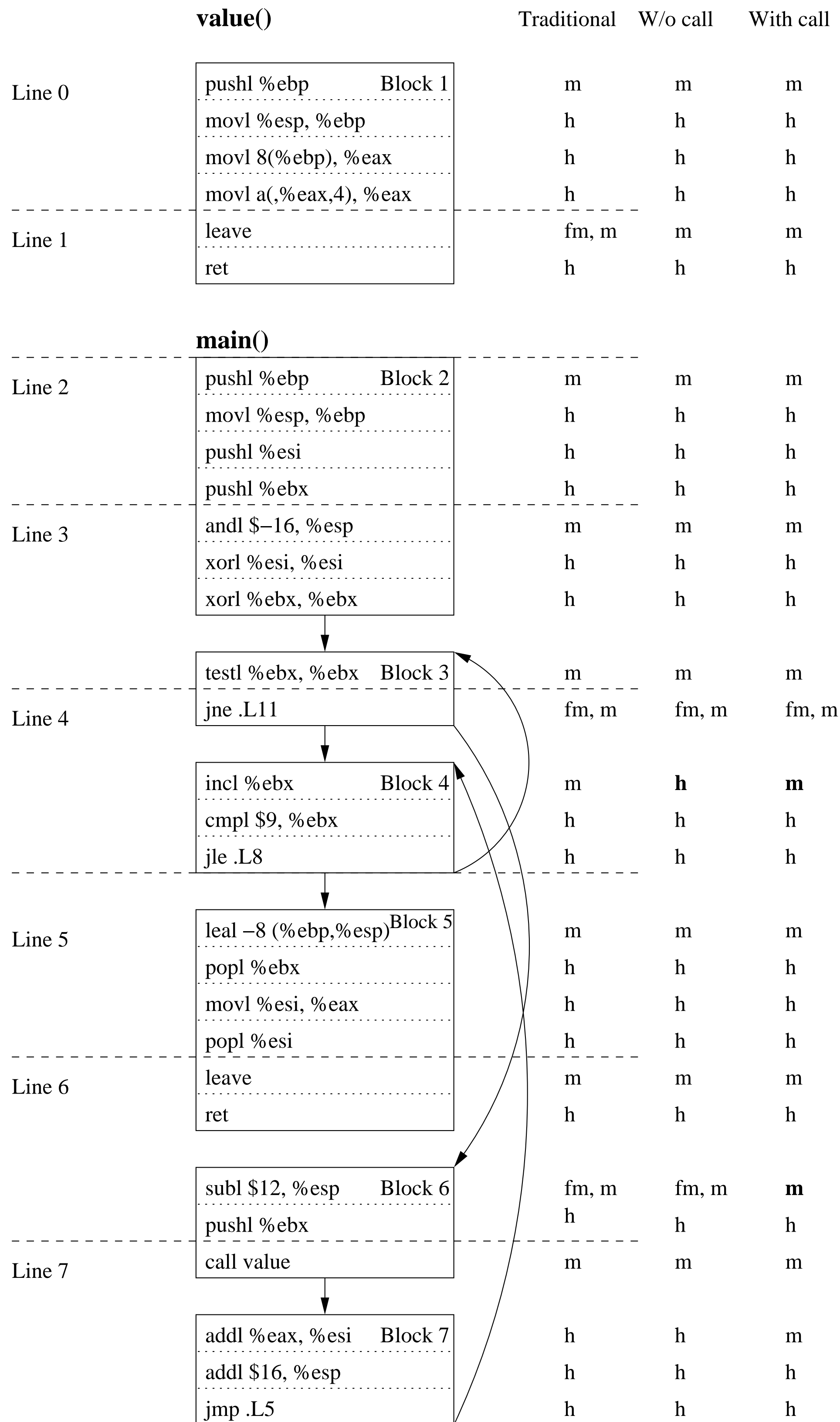


Figure 3.3: Compositional analysis example 3

Chapter 4

Experimental validation

This chapter describes the experiments performed to measure the performance improvement of the new compositional approach over the traditional approach and the accuracy of categorizations predicted by the new approach. Five benchmarks from the C-labs real-time benchmark suite [1] `adpcm`, `ndes`, `cnt`, `mm` and `fft` and one benchmark (`multimedia audio decoder - mad`) from MiBench [2] embedded benchmark suite were used.

4.1 Performance improvement

Time to perform static cache analysis using the traditional integrated approach and using the compositional approach were measured and compared for the six benchmarks for various cache configurations. Different cache configurations were used with the number of cache lines varying from 4 to 1024 lines (powers of 2) and size of each cache line varying from 8 bytes to 64 bytes.

In this section, graphs for three of the six benchmarks are shown, because the patterns in the results obtained for these three benchmarks are representative of the complete set. Figures 4.1, 4.3 and 4.5 show the comparison between time required by the traditional approach and time required by the compositional approach for a cache line size of 16 bytes. Figures 4.2, 4.4 and 4.6 depict results for a cache line size of 32 bytes. It is important to note that in these figures the time taken by only the second stage (compositional analysis) is considered. The first stage (module-level analysis) needs to be performed only once for each module for a particular cache configuration and, hence, it is not considered for com-

parison purposes. This is in line with the main purpose of this thesis, which is to transfer the computational burden from the compositional stage to the module-level stage.

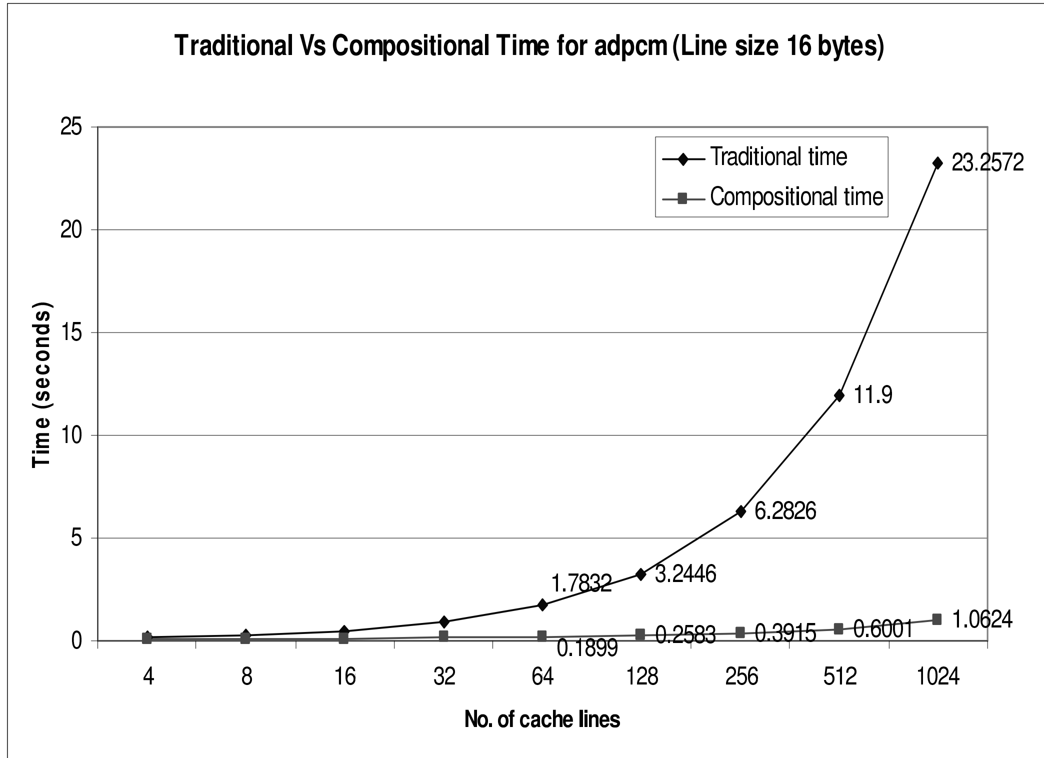


Figure 4.1: Comparison of computation times for `adpcm` for cache line size=16 bytes

For `adpcm` the compositional approach takes lesser time than the traditional approach for all the cache configurations. The figures for `adpcm` also indicate that the compositional approach scales better than the traditional approach. For instance, in Figure 4.1 the increase in traditional time for a change from 256 to 512 lines is about twice, whereas the increase for the compositional time is only about 1.5 times. Both Figures 4.1 and 4.2 show similar trend in the required computation time for the two approaches.

For `mm`, for a smaller number of cache lines (less than 128), the traditional approach performs better than the new approach, but for larger cache sizes, the new approach takes lesser time than the traditional approach. `mm` is a small benchmark (around 130 instructions). Its calling structure and the control structure are fairly simple. So it does

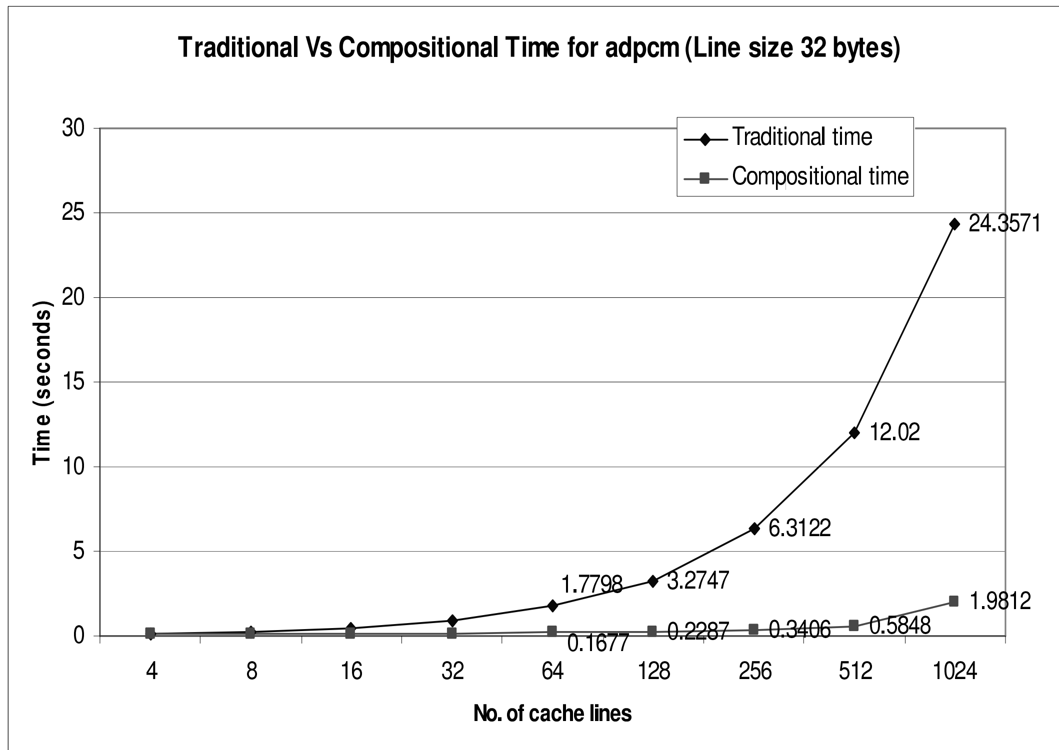


Figure 4.2: Comparison of computation times for `adpcm` for cache line size=32 bytes

not provide enough opportunity for the compositional approach to exploit any complexity in the program structure, but as the number of cache lines increases, the overhead in the traditional approach increases with higher rate and hence the new approach produces better results.

`mad` is a fairly large benchmark with around 9000 instructions. It consists of around 74 functions and the program structure involves complex loop and control-flow constructs. As is expected, there is a significantly large improvement in the compositional approach. For instance, the traditional approach takes about 1 hour for 512 lines compared to only about 32 seconds for the new approach. It should also be noted that for a cache line size of 16 bytes, more time is required than for 32 bytes cache lines.

Figures 4.7, 4.8 and 4.9 also compare the traditional and compositional times as in the previous figures, but they take both stages of the new approach into account while

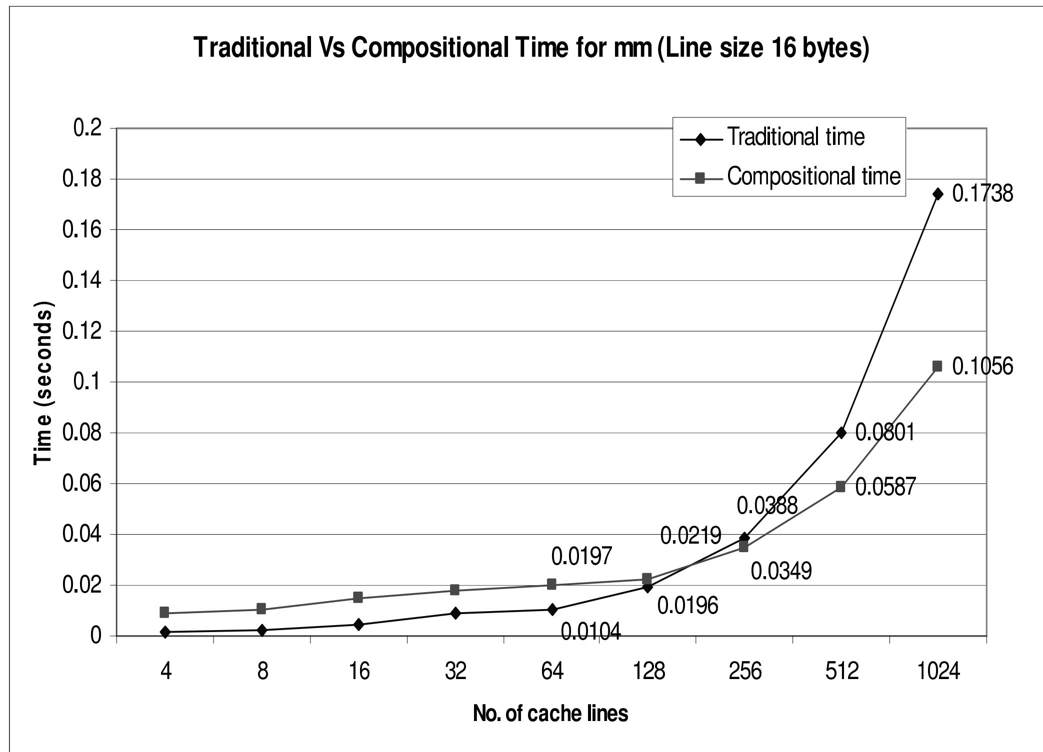


Figure 4.3: Comparison of computation times for `mm` for cache line size=16 bytes

calculating the compositional time.

Figure 4.7 shows that for `adpcm` the total time for the two-stage approach is lesser than that for the traditional approach for cache lines ranging from 4 to 512. For a cache with 1024 lines, the time for the modular stage exceeds the traditional time and this causes the total time to be larger for the new approach. The modular stage performs 4 different analyses on each module, hence, this overhead increases with the increase in cache size. This explains the behavior for a cache with 1024 lines.

For `mm` (Figure 4.8), the total time for the new approach is greater than the traditional time for all cache configurations. The reason for this remains the same *i.e.* the smaller size of `mm`.

For `mad` (Figure 4.9), a significant savings in computation time is observed for all different cache sizes. For a cache with 1024 lines, the new approach takes a total of

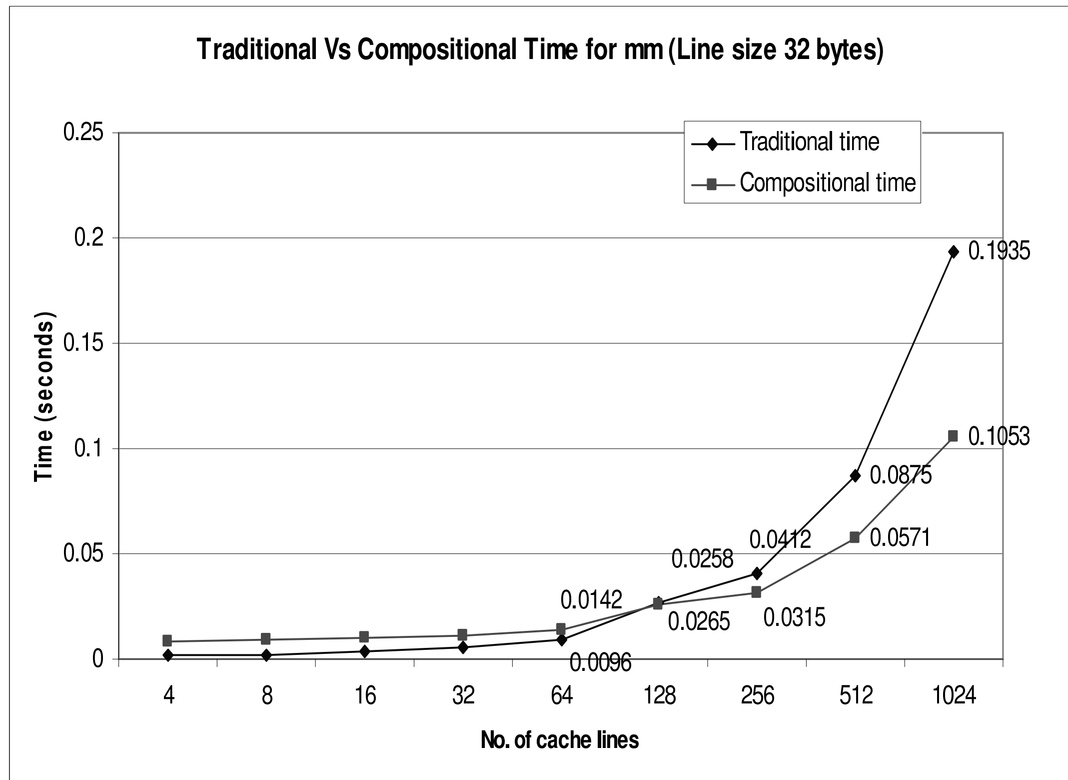


Figure 4.4: Comparison of computation times for `mm` for cache line size=32 bytes

around 350 seconds compared to the 6600 seconds of the traditional approach. Thus results for fairly larger benchmarks (`adpcm` and `mad`) show that the compositional approach, even when applied just once for all the modules in a program, performs much better than the traditional approach. This is significant because it indicates that for such large programs, the new approach can result in large savings in computation time.

4.2 Accuracy of predicted categorizations

4.2.1 Comparison of worst-case execution cycles

To determine the accuracy of predictions, worst-case execution cycles (WCEC) were measured for all the benchmarks using the timing analyzer discussed earlier. Data on WCEC were collected for categorizations predicted by the traditional approach as well as by

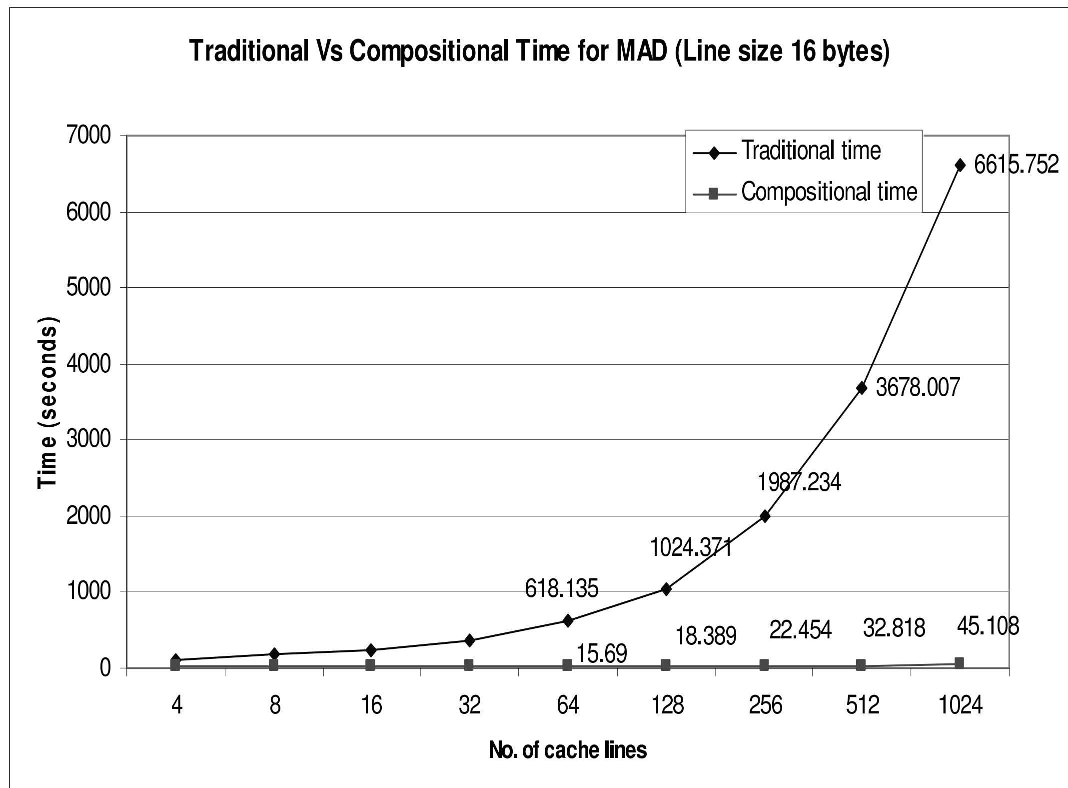


Figure 4.5: Comparison of computation times for `mad` for cache line size=16 bytes

the new approach. Then the WCEC obtained using the two approaches were compared. The timing analyzer was run to perform analysis for `Pseudo-Instruction Set Architecture` (`PISA`) [3]. To obtain the control-flow analysis information, the benchmarks were compiled using `gcc` for PISA (`pgcc`).

For all benchmarks except `adpcm`, for all cache configurations ranging from 4 cache lines to 512 cache lines, the WCEC were exactly the same. For `adpcm` and `ndes`, the result of the timing analysis are summarized in tables 4.1 and 4.2 respectively.

For `adpcm`, results for a cache size of 64 lines show a tighter WCEC prediction by the new approach. This is the only difference observed between WCEC for the two approaches. In this case, the traditional approach seems to categorize one instruction of `adpcm` pessimistically from a first-miss to a miss. The new approach predicts categories of this instruction more accurately. This leads to a tighter and still safe WCEC bound

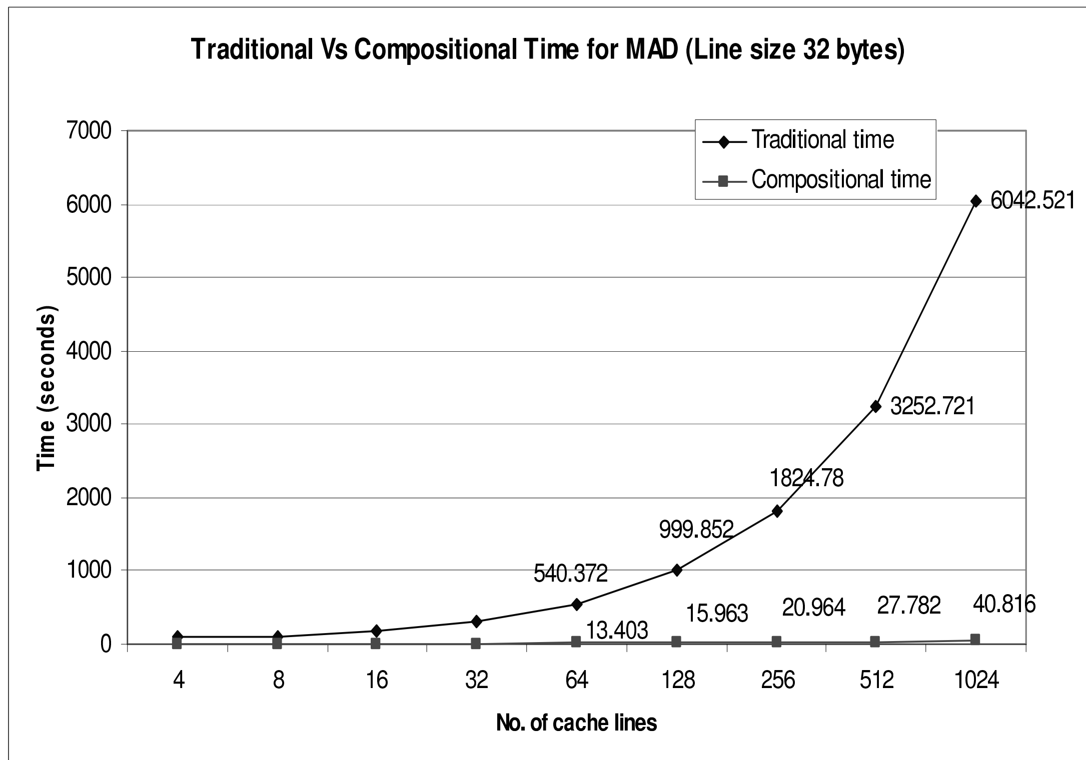


Figure 4.6: Comparison of computation times for `mad` for cache line size=32 bytes

using compositional approach. It is also interesting to note that as `adpcm` occupies around 230 program lines in memory, it entirely fits in caches with 256 and 512 lines. *I.e.*, once a program line is brought in cache, it will not be evicted later. Hence the WCEC for these two cache sizes are the same for this benchmark.

For `ndes`, as can be seen from the table, the estimated WCECs are exactly the same for the two approaches.

4.2.2 Comparison of WCEC with Simplescalar simulator

As shown in the above section, WCECs for the six benchmarks were obtained using the timing analyzer for the various cache configurations. These numbers were compared to the number of cycles obtained from the architectural simulator for the simplescalar toolset. The benchmarks were executed for worst-case input data on the simulator.

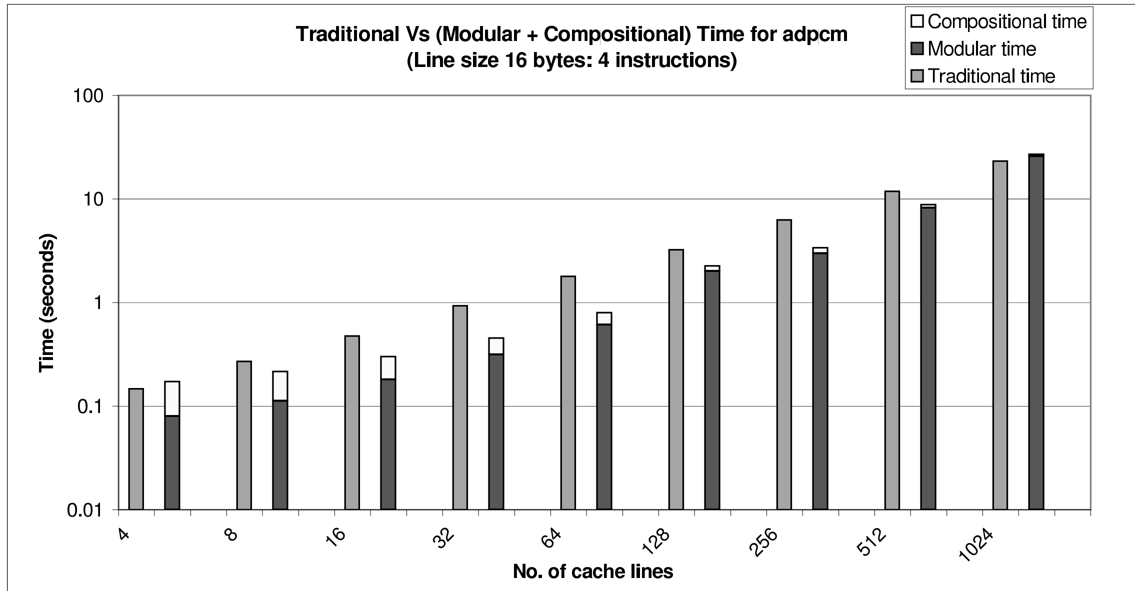


Figure 4.7: Comparison of total compositional time with traditional time for `adpcm`, cache line size=16 bytes

Number of cache lines	Traditional WCEC	Compositional WCEC	% change
512	9,227,240	9,227,240	0
256	9,227,240	9,227,240	0
128	18,590,441	18,590,441	0
64	23,751,141	23,651,041	-0.42

Table 4.1: Results of timing analysis of `adpcm`

Table 4.3 shows the WCEC comparison for a direct-mapped cache with 64 lines and a line size of 16 bytes. For other cache configurations, similar results are observed. The WCEC of `mad` could not be obtained due to constraints of path analysis within the timing analysis tool, a tool that is beyond the scope of this thesis.

4.3 Summary

The validity of the new approach is dependent on two metrics :

- the improvement in the computation time required to perform static cache analysis

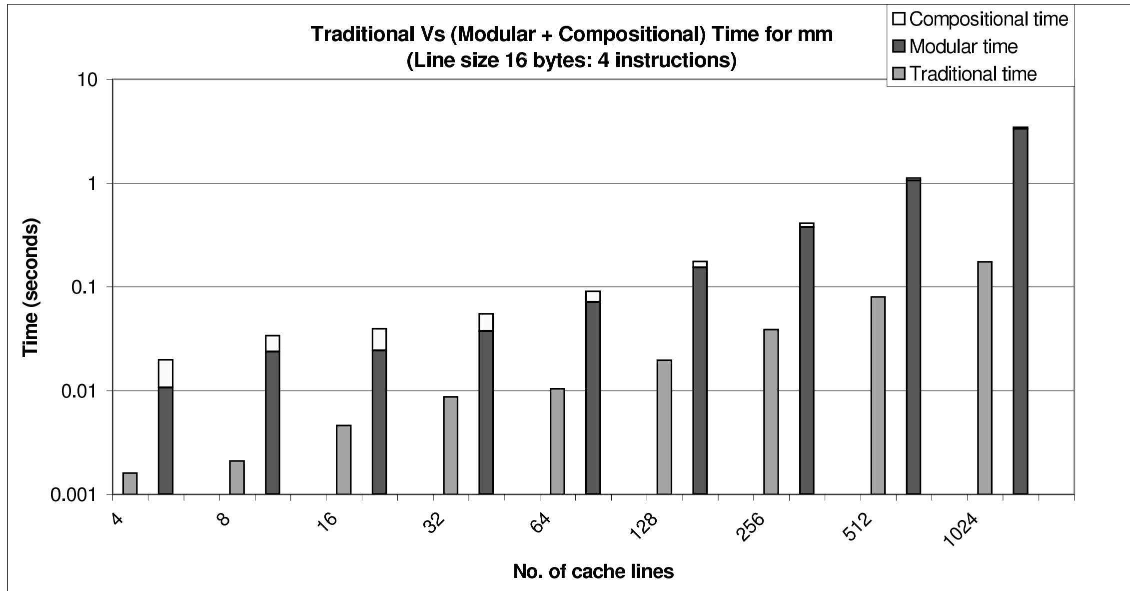


Figure 4.8: Comparison of total compositional time with traditional time for `mm`, cache line size=16 bytes

Number of cache lines	Traditional WCEC	Compositional WCEC	% change
512	131,514	131,514	0
64	146,583	146,583	0
16	347,523	347,523	0
8	839,551	839,551	0

Table 4.2: Results of timing analysis of `ndes`

- the accuracy of the predicted categorizations.

The compositional approach certainly leads to a significant improvement in execution time of static cache analysis. It also provides worst-case categorization predictions at least as accurate as those provided by the traditional approach. In some cases, it provides more accurate and still safe predictions than the traditional approach.

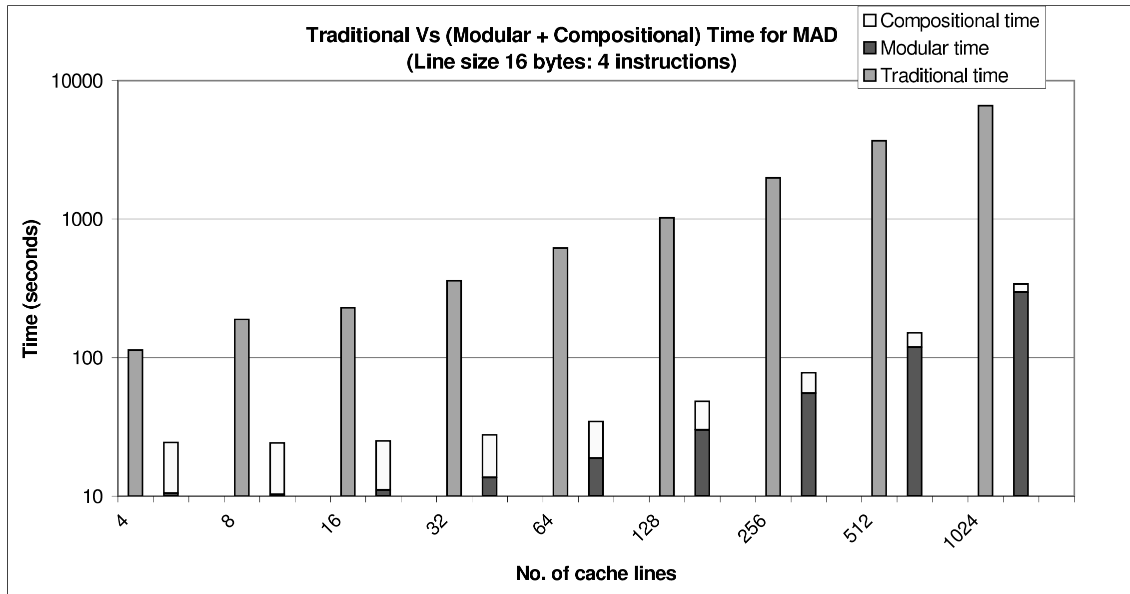


Figure 4.9: Comparison of total compositional time with traditional time for `mad`, cache line size=16 bytes

Benchmark	Predicted WCEC	Observed WCEC	Ratio
`adpcm`	23,740,141	17,549,967	1.35
`ndes`	146,298	95,147	1.54
`fft`	381,091	369,671	1.03
`cnt`	72,240	71,411	1.01
`mm`	2,037,588	2,034,313	1.01

Table 4.3: Comparison of predicted Vs observed WCEC

Chapter 5

Future Work

The basic compositional static cache prediction approach can be extended to incorporate more functionality. This is described in the following sections.

5.1 Support for set-associative caches

Currently, the compositional framework only supports direct-mapped instruction caches. The basic paradigm of this approach can be used for set-associative caches with some changes in the rules for adjusting the categorizations when a module acts as a callee and as a caller. For example, the incoming line calculations will change accordingly as each set can now have more than one cached lines. Also, the exit lines calculations for each module will change, and they need to be calculated for each set in the cache.

5.2 Best-case categorizations

As of now, the new framework predicts worst-case categorizations for each instruction reference. Using the definitions for best-case categorizations [18, 19] and with some changes to the current model, the best-case categorizations can be predicted. The calculations of incoming lines for each basic block and the calculations of exit lines for each module need to be changed to incorporate best-case scenarios when control-flow transitions cross module boundaries.

5.3 Multi-level modular analysis

Consider the case of *C string library* functions `strstr` and `strlen`. `strstr` has a call to `strlen`. Suppose the module-level analysis for `strlen` is performed and categorizations are stored. Now, while performing module-level analysis for `strstr`, instead of calculating effects of `strlen` as an unknown module, we can use the available module-level analysis for `strlen` to arrive at the adjusted categorizations in the module-level stage itself instead of delaying the adjustments to the compositional stage. This can certainly be done, especially when the called module (in this case `strlen`) is larger than the entire cache. Similarly, for `strlen`, while calculating categorizations for simulated outer-loops, we can use the module-level analysis of `strstr`. Again, this can certainly be done when the caller module `strstr` is larger than the entire cache. This work should be more effective in case of small caches and fairly large-sized modules. It reduces the complexity of the compositional stage even further.

Chapter 6

Related Work

This chapter describes prior work on cache analysis and timing analysis. First, the general concepts related to static cache simulation approaches and the existing frameworks that use these concepts are introduced. This is followed by a summary of the new compositional analysis framework.

6.1 Prior methods

The main purpose of the various static cache simulation methods is their use in obtaining worst-case execution time (WCET) for a program. Bounding WCET estimations is important as schedulability analysis for hard real-time systems requires that the WCET be known in order to ensure feasibility of scheduling a task set for a given scheduling policy, such as rate-monotone and earliest-deadline-first scheduling [17].

Many different approaches have been used to obtain the WCET for a program. The combined use of the static cache analysis (Arnold *et al.* [4]) and the static timing analysis (Healy *et al.* [6]) provide fairly accurate WCET bounds. Our approach splits the static cache analysis into two stages, reusing the results from the first (module-level analysis) stage in the second (compositional analysis) stage, thus addressing the issue of computation overhead of cache analysis.

Harmon *et al.* [5] describe a retargetable framework for predicting execution time that transforms the machine-level instructions into a sequence of primitive operations, which express the functionality of each instruction in fine-grain detail. This framework along with Puschner *et al.* [21] and Park [20] propose ideas for timing analysis of unoptimized programs

on simple CISC processors.

Zhang *et al.* [15] introduce an analyzable pipeline model that considers the effect of pipelined execution on timing analysis. The combined framework of [4] and [6] described above take instruction cache effects into account. These ideas and WCET analysis framework in [10] introduce timing analysis for pipelined RISC processors.

Li *et al.* [8] propose the use of Integer Linear Programming (ILP) formulation for capturing structural,functionality and cache constraints and solve them to obtain WCET. The ILP approach is extended to take cache timing effects into account. Rawat [16], Kim *et al.* [7], Li *et al.* [9] and White *et al.* [14] extend the conventional static timing analysis to incorporate data cache timing effects.

Some instruction-level dynamic timing analysis methods assume the knowledge of an input set that would trigger the worst-case path during the actual program execution. Finding such an input set can be a non-trivial problem involving very high computational complexity. Even if such an input set is found, the actual execution-time effects of architectural features onto timing analysis might cause a different input set to produce WCET.

The integrated path and timing analysis method in Lundqvist *et al.* [11] combines dynamic cache analysis and dynamic timing analysis using instruction-level cycle-accurate simulation technique. The cache state at various points in the timing analysis is calculated on the basis of the actual memory references made. The worst-case state is obtained by analyzing the cache state that will trigger worst-case response time in the remaining execution sequence. Thus, the cache calculations are integrated in the instruction-level simulation. Such approaches would suffer a significant computational overhead involved in cache calculations for fairly large programs that consist of a large number of possible execution paths.

Vera *et al.* [13] describe a technique of data cache locking to achieve predictable worst-case execution time for a program even when the execution environment has data caches. Unpredictability in WCET estimations is eliminated by locking those regions in the code that cannot be analyzed statically. Performance degradation possible due to cache locking is reduced by loading the cache with data likely to be accessed, but some overhead prevails.

From this review of prior work, it is evident that all these ideas deal with the problem of accurately estimating WCET for a program in the presence of architectural features, such as data and instruction caches and instruction pipelines. Our work is different

from these as it focuses on the computation overhead involved in static cache analysis (performed separately from timing analysis), while preserving the accuracy of the predicted categorizations.

Rubin *et al.* [12] discusses a profile-analysis framework for data-layout optimizations. It uses the concept of deriving a *data objects trace* to save the re-execution of programs to assess the performance of a newly computed data layout. Each entry in the *data objects trace* contains a symbolic address as well as a unique identifier of the referenced data object. To evaluate a candidate data layout, each data object is assigned its new memory location obtained from the layout. Then, the original data trace is simulated using the *data objects trace* and the new memory addresses to measure the new memory behavior. Thus, the *data objects trace* parameterizes the actual memory address trace, making its reuse possible. Since the *data objects trace* is address-independent, it can be reused in the new memory layout evaluation. In this way, the idea of address parameterization is utilized here, but it is for a very different goal of data layout optimization. Our work also parameterizes the data-flow information obtained in the first stage (module-level analysis) as the absolute address information is not available during this stage and the inter-procedural analysis on modules belonging to a program can be performed irrespective of the actual absolute addresses the modules are mapped to.

Chapter 7

Summary

This thesis contributes an approach to reduce the computation overhead involved in obtaining cache hit/miss information from static cache simulation for direct-mapped instruction caches. Categorizations obtained in module-level analysis are reused in compositional analysis that performs the necessary inter-procedural analysis and derives the final categorizations. Thus for any particular cache configuration and for any module, the module-level analysis needs to be performed only once. The compositional stage can reuse this analysis for any program that contains the module.

The experimental results show that the compositional approach performs much better than the traditional approach for large cache sizes and fairly large benchmarks. For smaller cache sizes the computational overhead is comparable to the traditional approach. Compositional approach also gives predictions at least as accurate as the traditional static cache simulation approach. This was evident from the worst-case execution cycles obtained for the benchmarks for the two approaches. This work provides an efficient way for analyzing direct-mapped instruction caches statically especially for large cache sizes and for programs with a complex loop structure and a complex calling structure.

Bibliography

[1] C-lab wcet benchmarks. http://www.c-lab.de/home/en/download.html.

[2] Mibench embedded benchmark suite. http://www.eecs.umich.edu/mibench.

[3] Portable instruction set architecture. http://www.simplescalar.com/overview.

[4] Arnold et al. Bounding worst-case instruction cache performance. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 172–181, Raleigh, North Carolina, 1994.

[5] Harmon et al. A retargetable tehnique for predicting execution time. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 68–77, 1992.

[6] Healy et al. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of sixteenth Real-Time Systems Symposium*, pages 288–297, Raleigh, North Carolina, 1995.

[7] Kim et al. Efficient worst case timing analysis for data caching. In *Proceedings of IEEE Real-Time Technology and Applications Symposium*, 1996.

[8] Li et al. Efficient microarchitecture modelling and path analysis for real-time software. In *Proceedings of sixteenth Real-Time Systems Symposium*, pages 298–307, 1995.

[9] Li et al. Cache modelling for real-time software: Beyond direct mapped instruction caches. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 254–263, December 1996.

[10] Lim et al. An accurate worst case timing analysis for risc processors. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 97–108, December 1994.

[11] Lundqvist et al. An integrated path and timing analysis method for cycle level symbolic execution. In *Proceedings of Real-Time Systems Symposium,1999*, pages 183–207, 1995.

[12] Rubin et al. An efficient profile-analysis framework for data-layout optimizations. In *Proceedings of 29th Annual Symposium on Principles of Programming Languages*, 2002.

[13] Vera et al. Data cache locking for higher program predictability. In *Proceedings of SIGMETRICS 2003*, June 2003.

[14] White et al. Timing analysis for data and wrap-around fill caches. *Real-Time Systems*, 17(2/3):209–233, November 1999.

[15] Zhang et al. Pipelined processors and worst case execution times. *Real-Time Systems*, 5(4):319–343, October 1993.

[16] J.Rawat. Static analysis of cache performance for real-time programming. Master's thesis, Iowa State University, 1995.

[17] Liu and Layland. Scheduling algorithms for multi-programming in a hard real-time environment. *J. of the Association of the Computing Machinery*, 20(1):46–61, January 1973.

[18] Frank Mueller. *Static Cache Simulation and its Applications*. PhD thesis, Florida State University, Department of Computer Science, Tallahassee, Florida, July 1994.

[19] Frank Mueller. Timing analysis for instruction caches. *Real-Time Systems Journal*, 18(2/3):209–239, May 2000.

[20] C.Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–61, March 1993.

[21] Puschner and Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, September 1989.