Worst case timing analysis of input dependent data cache behavior

Jan Staschulat, Rolf Ernst
Institute of Computer and Communication Network Engineering
Hans Sommer Str. 66, D-38106 Braunschweig, Germany
{staschulat|ernst}@ida.ing.tu-bs.de

Abstract

Data caches significantly reduce the average memory access time and are necessary for an efficient design. Due to its direct dependency on input data is is difficult to predict the worst case timing behavior, which is crucial for a reliable system. While simulation is too time-consuming, current worst case execution time approaches focus on instruction caches only. Current approaches to data cache analysis restrict cache behavior to predictable data accesses or classify input dependent memory accesses as non-cacheable.

In this paper we propose a worst case timing analysis for direct mapped data caches that classifies memory accesses as predictable or unpredictable. For unpredictable memory accesses, a novel analysis framework is proposed that tightly bounds the impact on the existing cache contents as well as cache behavior of unpredictable memory accesses themselves. For predictable memory accesses, we use a local cache simulation and data flow techniques. Furthermore, we describe an implementation of the analysis framework. Several experiments demonstrate its applicability. The approach targets real-time software verification but is also useful for design space exploration.

1 Introduction

While processor speed is steadily increasing, background memory access time remains slow. Caches can significantly reduce the average memory access time, leading to a total shorter execution time. Timing behavior is becoming a serious problem in many embedded systems. In soft and hard real-time systems, timing guarantees are necessary to verify the functional behavior as well as to efficiently use hardware resources.

Current practice is cache simulation to determine the typical timing behavior, which is unsafe because not all program paths can be covered. A full coverage would require an exponential number of test data and would be too time consuming. Therefore, only a subset of all program paths is tested. An alternative approach is static timing analysis that delivers safe bounds of the worst case execution time. In the last decade, many techniques have been proposed for tasks running on a single processor architecture with a complex design, including pipelines [19], caches [14], and branch predictors [2].

There are several approaches to make caches more predictable and efficient. One approach is to partition the cache sets and to reserve these partitions for individual tasks [12]. The advantage is that cache lines do not have to be reloaded after interrupts and between consecutive executions of the same task. Also, cache behavior becomes (partly) orthogonal for tasks and therefore more predictable. Task layout techniques are suggested in [4], which aim at minimizing the inter-task interference in the instruction cache. Another approach is to lock frequently used cache blocks. Such techniques have been investigated in [9] [3] [17]. These approaches increase area and power cost as they require larger caches and background memory to become effective. Therefore, heterogeneous memory architectures with caches and scratch-pad SRAM have been introduced, like TriCore, where the scratch-pad can hold frequently used cache blocks. Compiler techniques for such architectures have been proposed by [16]. While cache partition and lock strategies are certainly a very useful add-on to improve cache predictability and efficiency, they do not solve the general cache analysis problem in which all tasks share the entire cache.

For a single task execution, the timing behavior for instruction caches has been extensively studied [14] [24]. However, data cache behavior has often been restricted in static worst case timing analysis. Previous approaches have focused on predictable memory access pattern, for example [6] [21] [18], or have simplified the timing behavior for input dependent memory accesses [13] [5].

The contribution of this paper is a worst case timing analysis for data caches that classifies memory accesses as predictable and unpredictable. For unpredictable memory accesses we propose a novel timing analysis framework that tightly bounds the impact on the existing cache contents as

Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS'06)
0-7695-2619-5 /06 $20.00 © 2006 IEEE

well as the number of cache misses for unpredictable memory accesses itself. For predictable memory accesses, we propose a local cache simulation and data-flow techniques. As a second contribution, we describe an implementation of the analysis framework. Finally, we compare the new approach with a previous approach and cache simulation.

The key benefit of the proposed approach is a retargetable analysis framework to verify the timing behavior of data caches that considers input dependent memory accesses. The framework can be used for design space exploration as well as system level optimization.

1.1 Related Work

In early work by [8] a heuristic argument of the *pigeon-principle* is proposed for direct mapped data caches. We explain the principle by an example where a memory access to an array occurs in a loop with a maximum of 100 iterations. Supposing that the array has ten elements, it can be concluded that 10 cache misses and 90 cache hits will occur given the following assumptions: 1.) the entire array fits in the cache, 2.) the cache is direct mapped and 3.) none of the array elements are replaced during program execution. Because none of the array elements are replaced there can be at most 10 (compulsory) cache misses all other have to be cache hits.

An integer linear problem (ILP) formulation was proposed by [11] for direct mapped data caches. A cache conflict graph is constructed in which each memory address corresponds to a variable in the equation system. Because the complexity scales with the data size of arrays, is is unclear how efficient this approach will be for larger arrays. The dependency of index variables and array ranges are assumed to be given. This restricts the application domain to fully predictable array accesses. Unfortunately, no results from experiments are presented.

Abstract interpretation is used to predict data cache behavior in [5]. A persistence analysis determines the maximum number of cache blocks that remain in cache. This is used to calculate a lower bound on cache hits. Value analysis is used to calculate the possible range of array accesses. For an unpredictable array access, it is assumed that all cache blocks of an array are accessed, which replace many other cache blocks. This means, that a single access to an unpredictable array is modeled as if all elements of the array would have been accessed. However, only one cache block can be replaced. The method is only described in theory without providing experimental results.

Data flow analysis techniques have been applied to data cache analysis by [22]. The address range of data references is performed on low-level representation after code generation and all optimizations. Unknown data references are not considered and array ranges would have to be annotated by the user.

A symbolic simulation technique is proposed by [13] to detect predictable and unpredictable memory accesses. Unpredictable data structures are tagged as non-cacheable and consequently always require a cache miss. The main drawback is that input dependent memory accesses are classified as non-cacheable and are treated as always cache miss.

A different approach is to use cache miss equations(CME). Introduced by [6] and developed by [21], these CME compute re-use vectors to calculate cache accesses within loops. It is assumed that all array accesses are affine functions of loop induction variables and that loops are perfectly nested. In [18] the CME-framework is extended to consider scalar variables and to allow more sophisticated loop structures. However, the CME frameworks always assume that array index variables do not depend on input data.

1.2 Principle of analysis

The main open problem in data cache analysis is the timing behavior of unpredictable memory accesses. Trace-based simulations would be too time-consuming to cover the worst case. Existing data-cache frameworks exclude input dependency, such as the CME frameworks [6][21][18], or classify input dependent memory accesses as non-cacheable [13]. In [5] the impact of unknown data accesses on the cache contents is simplified by assuming that all elements of an array are loaded to the cache for each array access. However, only one array element will be loaded to the cache.

To reduce this overestimation, we classify array accesses as predictable and unpredictable memory accesses, if the index expression can be statically computed or not. Each scalar variable is predictable, because the memory location is fixed. An unpredictable memory access has two influences on cache behavior. First, it has an impact on the current cache contents by possibly replacing some other cache block. Secondly, the access itself requires an additional cache miss, if the element is not in the cache.

For the first impact, a cache miss counter C_m is defined for each unpredictable memory access. The counter represents possible replacements of some cache blocks. We refine this idea, by incrementing C_m only if a useful cache block can be replaced by some element of the array. A useful cache block is a cache block that contains a data variable that is used a second time after the unpredictable access occurred. Only these cache blocks will require an additional cache miss in the future execution.

For the second impact, we bound the number of cache misses by comparing the array size and the execution counts as suggested in [8]. The key difference in this approach is the computation of the set of persistent cache blocks. In [8] no memory blocks are allowed to be replaced. We apply the

pigeon-hole principle to those arrays whose elements are all persistent (e.g. are never replaced by other memory blocks).

For all predictable memory accesses we use local cache simulation and global data flow analysis. Finally, we set up an integer linear problem (ILP) to consider predictable and unpredictable memory accesses. Additional constraints for the pigeon-hole principle are added to the ILP formulation.

1.3 Framework overview

The analysis framework is shown schematically in figure 1. The analysis is based on the source code of an application. An intermediate representation, the abstract syntax tree, is generated from the source code. Based on this representation, data dependency analysis identifies single data sequences and a control flow graph is constructed (section 2). In a second step, the actual memory addresses are extracted using the linked object code and are mapped to the control flow graph (section 3). Then follows a cache behavior analysis for predictable and unpredictable data accesses (section 4) and, finally, the timing analysis computes the worst case program execution path (section 5).

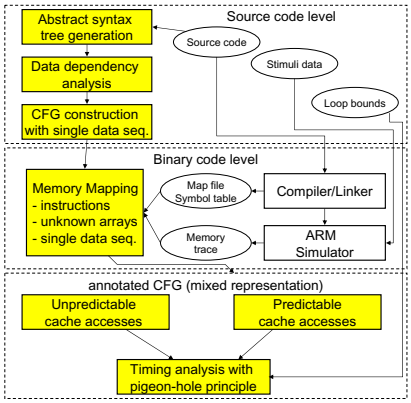

Figure 1. Overview of analysis framework.

Experimental results are provided in section 6. and section 7 concludes the paper.

2 Data dependency analysis

Data cache behavior has been simplified in worst case timing analysis because of unpredictable data accesses. A single instruction might access different memory locations during the execution of a program. In such cases, it is already difficult to predict which memory locations are accessed, and it becomes very complicated to decide whether such accesses are cache hits or misses. However, sometimes memory locations, for example array elements, are accessed in a predictable order. Two conditions are necessary: first, an input data independent control flow and, secondly, input data independent memory accesses.

We say that a program segment contains a *single data sequence* (SDS) if it contains only input independent control structures and input independent memory accesses. A program segment is a sequence of basic blocks, in which basic blocks are the smallest entity of a program [1]. An example source code is shown in figure 2. The `for`-loop in lines 11-12 contains such a SDS, because the condition $i < 100$ does not depend on input (or global data) and the index variable of $b[i]$ does not depend on input data. On the other hand, the program segment in lines 6-10 do not contain a SDS, because the condition $a < 2$ depends on global variable a. Also, the array access in line 9 depends on the global variable a.

```
 1 int a;      // global variable
 2 int b[100]; // global array
 3 int main(){
 4     int i;
 5     int k;
 6     if (a<2){
 7      k = 10;
 8     } else {
 9      b[a] = 0;  // input dependent access
10     }
11     for (i=0;i<100;i++){
12      b[i] = i;  // input independet access
13     }
14     return 0;
15 }
```

Figure 2. Source code example.

In this section we present a methodology to identify SDSes. First, we discuss the proper analysis level in section 2.1. Then, we describe how to identify input dependent control structures in section 2.2 and input dependent memory accesses in section 2.3. Finally, we construct a control flow graph with SDS in section 2.4.

2.1 Analysis abstraction level

Single data sequences require input independent control structures and memory accesses. These properties can be computed by data dependency analysis which could be performed either on source code or on object code level. A source code level analysis would be based on a control flow graph, where nodes contain assignment statements in high level language. The drawback is that actual memory addresses cannot be derived because instruction set of the assembly language, linking information, compiler options and memory layout are not available. The exact memory mapping has to be derived in a second step from the actual binary by disassembling the object code. For a correct source code correspondence, optimizations beyond ba-

sic block level are not allowed. An approach to analyze input-dependency for high level programming language has been proposed by [7].

As an alternative, data dependency analysis could be executed on object code level where the actual memory layout information is available. From the object code a control flow graph can be generated, as suggested in [5][22][11]. However, data dependence analysis is difficult, because of indirect register addressing and the dependence on the instruction set. For example the ARM instruction set contains conditional instructions, which makes value analysis complicated. Secondly, such an analysis is limited to the analyzed instruction set only, for every new processor a new data dependency module would be necessary.

We chose the first alternative, in which the data dependency analysis is done on source code level because it is hardware independent and value analysis is simpler than on object code level. The lack of actual memory addresses is resolved by annotating the control flow graph with the memory addresses using cache simulation in a second step (section 3.

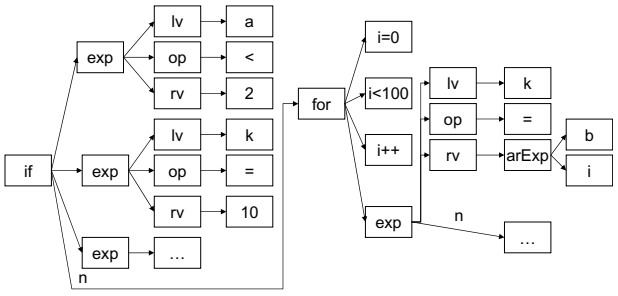

Figure 3. Abstract syntax tree.

For data dependency analysis, an abstract syntax tree (AST) [1] is generated from the source code program. The abstract syntax tree is used because it keeps direct correspondence between program segments and the control flow hierarchy [24]. An AST contains hierarchical nodes representing statements and edges that represent the hierarchy of statements. For a part of the source code of figure 2 a simplified abstract syntax tree is shown in figure 3. Edges connect a node with a lower-hierarchy node, for example, to a `then`, an `else` expression, or connect a node to the next node (n) on same hierarchy level. The `for` statement has edges to the init-, condition-, body-, update and next-expression. An assignment expression contains a left value (`lv`), operator (`op`) and a right value (`rv`). An array expression (`arExp`) has two sub-expressions: the name of the array (`b`) and the index expression (`i`).

2.2 Program path classification

The execution path of a program depends on input data which makes it difficult to predict precise worst case timing bounds. But often a program contains only a single feasible program path. An example is a fast Fourier transformation (FFT) or a FIR filter algorithm. In such structures the control flow only depends on the internal state, but not on input data. For example, a loop with several `if-then-else` statements that only depend on the loop iteration variable. In this case we can use trace-based simulation. In previous work we have presented a methodology to identify single feasible paths (SFP)[23] and proposed a timing analysis for instruction caches [24].

A SFP is determined by analyzing the conditions of each control structure in the abstract syntax tree. A control structure that does not contain other hierarchical control structures is called a SFP, if the condition does not depend on input data, otherwise it is called a multiple feasible path (MFP). A control structure with an input independent condition that contains substructures with MFPs is also classified as a MFP.

The input dependency of conditions is determined by a symbolic simulation algorithm on the abstract syntax tree. Each variable is either classified as input dependent or input independent. As the algorithm traverses the tree, the left value of an assignment is classified as input dependent if at least one variable on the right value is input dependent. A symbol table stores the classification of each variable. If a node has several predecessors like a `then` or a `else` branch, a variable is also classified as unpredictable if it contains different values in the symbol tables of the predecessor nodes. Loops are analyzed twice to propagate assignment-statements to higher level hierarchic nodes.

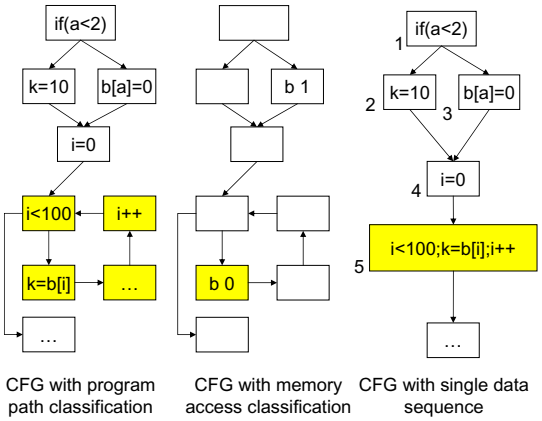

Figure 4. CFG in different analysis steps.

A control structure is input dependent if it contains at least one input dependent variable in the condition, other-

wise it is classified as input independent, and therefore a SFP. Figure 4 shows a control flow graph (CFG) on the left side, where the loop nodes are a SFP, because the loop condition does not depend on input data.

2.3 Memory access classification

The prediction of memory accesses for a data cache is more difficult than for instruction cache behavior because a single instruction may access a range of memory locations. To detect single data sequences, data cache memory accesses are classified as predictable and unpredictable. Memory accesses by scalar variables and predefined array accesses are classified as predictable. Otherwise, accesses are classified as unpredictable.

In section 2.2 we described a symbolic simulation algorithm on the abstract syntax tree to determine the input dependency of control structures. A similar algorithm can be used to determine the input dependency of memory accesses.

Initially, all scalar variables are classified as predictable, because the memory address is constant. Then the analysis proceeds as described above, propagating the input classification of each variable in the abstract syntax tree.

Instead of using the condition expression, the index expression of the array access expression determines if the memory location is input dependent or not. We define a memory access by an array as predictable if all variables of the index expression are input independent, otherwise it is classified as not predictable.

For example, the array expression in line 9 is unpredictable, but the one in line 12 is predictable. The output of the data dependency analysis is a CFG constructed from AST which is shown in figure 4 (middle). For each array variable a boolean variable signals the data dependency. For example, the assignment `b[a]=0` is data dependent (`b 1`) but the assignment `k=b[i]` is not.

For each node a tree traversal algorithm on the AST identifies whether an array access is predictable or not. Furthermore, the c-lines of each expression are annotated.

2.4 Single data sequence construction

A single data sequence (SDS) is defined by the program path classification (section 2.2) and memory access classification (section 2.3): A single feasible path that contains only predictable data structures is a SDS. An example of a SDS is shown in figure 4 (right).

At the end of this analysis step, a control flow graph is available where nodes contain single data sequences or unpredictable array accesses. In each node it is specified whether an array access is predictable or not. Our implemented algorithm stores the c-lines as well, but for a better understanding of the methodology we omitted the c-lines in this presentation.

3 Memory address mapping

In section 2, single data sequences have been constructed based on source code. The actual memory addresses are necessary for a data cache analysis. In this section, we describe how memory addresses are computed.

The methodology consists of three steps. First, instruction addresses are mapped to the CFG (section 3.1). Second, the memory access trace of instruction and data addresses is generated by a processor simulator (section 3.2). And finally, the data accesses of the memory trace are mapped to the nodes of the CFG (section 3.3).

3.1 Instruction address mapping

In the first step, instruction addresses are mapped to the control flow graph. A `map`-file is constructed that contains a table of memory addresses of assembly instructions and the corresponding c-lines. This table assumes that compiler optimizations beyond basic block boundaries are switched off. With a debugging tool, e.g. `addr2line`[1], the correspondence of memory addresses to c-lines can be computed. Then, instruction addresses are mapped those nodes in the control flow graph which have the same c-line. Figure 5 shows a part of the map file for the example source code.

```
address  c-line      address  c-line
80a8     9           80e8     11
80ac     9           80ec     12
80b0     9           80f0     12
80b4     9           80f4     13
80b8     10           ...
...
```

Figure 5. Memory map file.

3.2 Memory trace generation

Trace generation is used to collect the addresses of the predictable data accesses. Because the addresses are constant, profiling with branch coverage is sufficient. The memory trace of a software program is computed by an off-the-shelf processor simulator. For this study, we use the RealView Developer Suite[2], which is a cycle accurate processor simulator. In general, it suffices to use a functional simulator that outputs the memory trace. The simulator comes with the option to trace instruction and data addresses.

[1] Binutils tool suite http://www.gnu.org/software/binutils/

[2] http://www.arm.com/support/ARMulator.html

```
IT 000080A8 e59f0050 LDR   r0,0x8100
MNR4____ 00008100 00008414
IT 000080AC e5900000 LDR   r0,[r0,#0]
MNR4____ 00008414 00000000
IT 000080B0 e3500002 CMP   r0,#2
IS 000080B4 aa000001 bge   0x80c0
IT 000080B8 e3a0200a MOV      r2,#0xa
...
IT 000080EC e59f0010 LDR   r0,0x8104
MNR4____ 00008104 00008418
IT 000080F0 e7801101 STR   r1,[r0,r1,LSL #2]
MNW4____ 00008418 00000000
...
IT 000080EC e59f0010 LDR   r0,0x8104
MNR4____ 00008104 00008418
IT 000080F0 e7801101 STR   r1,[r0,r1,LSL #2]
MNW4____ 0000841C 00000001
...
```

Figure 6. Memory access trace.

An example of a memory access trace is shown in figure 6. The first letters denote the memory access type: `IT/IS` denotes instruction taken/skipped, `MNR4` denotes a memory access read (4 byte) and `MNW4` denotes a memory access write (4byte). The second column states the address fetched, the third the hex code and the rest of the line is the mnemonic assembly instruction.

Sufficient test patterns for a complete branch coverage are necessary. Branch coverage of a program means that each branch is executed taken and not-taken. The number of test patterns of a full branch coverage scales linearly with the number of branches while a full path coverage requires exponential number of test data.

3.3 Data address mapping

The objective is to map data accesses to control flow nodes. The instruction addresses were mapped to the nodes and can be used as identifiers. If one or more data addresses follow an instruction address d_i in the memory trace, then these data addresses are mapped to the nodes that contains d_i. We assume an in-order execution model of the processor. Otherwise, it would be more difficult to identify the instruction requesting a data address. For predictable data structures, especially in single data sequences, all data accesses are mapped to the corresponding nodes.

For unpredictable memory addresses this is not possible because a different test pattern would result in a different access pattern. Therefore, we have to assume that all memory locations are possible. The symbol table of the disassembled binary contains the start address and the size of each variable, for example the array `b[]` has the start address `8418` and a size of `0x190h`. This information is used to compute the range of possible memory accesses for unpredictable data accesses and is mapped to the corresponding nodes.

Table 1 summarizes the address mapping of instruction as well as data accesses for two nodes. The last column states whether the node is a single data sequence(SDS) or not. Note that the unpredictable array access in B_3 is annotated by its start address and its size. For the SDS in B_5, the exact simulation trace is used. Because of space restrictions, only a small part is shown and the array access is annotated with the actual sequence of data addresses.

Table 1. Memory address mapping summary

B_{id}	c−line	I-address	D-address	SDS
3	9	80a8, 80ac 80b0 80b4	8100 8414 **8418/0x190**	no
5	11,12	80EC 80F0 $\cdots$ 80EC 80F0	8104 8418 $\cdots$ 8104 841C	yes

4 Cache behavior analysis

The key idea of our approach is the observation that an access to an unknown data structure can remove at most one cache block. While the previous approach by [5] assumes that all cache blocks of an unknown data structure are assumed to be loaded to cache, we present a novel method to overcome this pessimism.

It consists of three steps. First, the analysis of the impact of unpredictable data accesses on existing cache content (section 4.1), second, the analysis of data cache behavior of unpredictable memory accesses themselves (section 4.2), and finally, the analysis of predictable cache accesses (section 4.3).

4.1 Cache miss counter

In this section, we describe the influence of an unpredictable data access on existing cache content. Assuming that an array has the size of ten cache blocks, any of these ten cache blocks might be requested by an access to this array. Which cache blocks of the current cache contents could be possibly replaced? The approach by [5] assumes conservatively that all ten cache blocks are loaded to the cache and (possibly) replace ten existing cache blocks. However, at most *one* cache block is actually removed, but we cannot statically predict which one.

To overcome such overestimation, a miss counter $C_m(B_i)$ is defined at node B_i. Initially, the miss counter is zero. Whenever a cache access to an unknown data structure occurs, the miss counter is incremented by one. This miss counter represents the number of *additional* cache misses for (possibly) replaced cache blocks. Note that this miss counter does not represent the number of cache misses for

unpredictable data accesses themselves. This will be described in section 4.2.

We give two algorithms to compute the cache miss counter. First, we could simply assume that `each` access to an unknown data address replaces a cache block. This is formalized in equation 1.

$$C_m(B_i) = |\{d| \quad \forall d \in Data(B_i)\}| \tag{1}$$

In other words, it is the cardinality of the set of unpredictable data accesses of node B_i. The set of data accesses is given by $Data(B_i)$ for basic block B_i. However, this would be too conservative. An additional cache miss is only required if a *useful* cache block is replaced. Useful cache blocks has been used by [10] [15] [20] to calculate the cache related preemption delay for fixed priority preemptive scheduling. A cache block is called useful if it may be available in basic block B_i (e.g. reaches this node) and may be accessed a second time via some outgoing path of B (e.g. is a live cache block). Two data flow analysis algorithms are set up, one to compute the reaching cache blocks $RCB(B_i)$ and one to compute the live cache blocks $LCB(B_i)$ at node B_i. The intersection of these sets is the set of useful cache blocks $UCB(B_i)$. The concept of data flow algorithms are further explained in section 4.2.

We refine the calculation of additional cache misses by incrementing the miss counter $C_m(B_i)$ only if there exists at least one useful cache block in the range of an unpredictable data access. This is shown in Equation 2:

$$C_m(B_i) = |\{d|UCB(B_i) \cap range(d) \neq \emptyset \forall d \in Data(B_i)\}| \tag{2}$$

where $range(d)$ is defined as the set of addresses of array d: $\{address(d), \cdots, address(d) + size(d) - 1\}$.

4.2 Persistence analysis of unpredictable data accesses

In case memory access are input dependent, heuristic arguments can be applied in a safe way. With the *pigeon-hole* principle, as described by [8], we can reduce the number of cache misses even though very little is known about data cache accesses. However the three assumptions are very restrictive. In our novel analysis framework we also assume that the cache is direct mapped and that the entire array fits in the cache. These restrictions seem feasible. But the third assumption is too restrictive.

We will statically analyze for which array addresses are never replaced by a persistence analysis. This persistence analysis checks that all elements of an array are still in the cache for all possible execution traces. The number of persistent cache blocks has been calculated in the approach by [5] with the conservative assumption that all cache blocks of unpredictable data structure are loaded to the cache. We use a similar technique, but with the goal to apply the pigeonhole principle in section 5.2.

Persistent cache blocks can be computed by data flow algorithms [1]. These data flow algorithms have been used in preemption delay analysis by [10] [15] [20]. Given a control flow graph, and some property P, a data flow algorithm propagates the property P to all nodes.

For persistence analysis, as described in [5], we define P as the set of reaching cache blocks. An iterative algorithm is used to solve the following equations:

$$P_{in}[B_i] = \bigcap_{P \in pred(B_i)} P_{out}[P]) \tag{3}$$

$$P_{out}[B_i] = \{r \odot gen[B_i] | r \in P_{in}[B_i]\} \tag{4}$$

$$c \odot c' = \begin{cases} c' & if \quad c' \neq \emptyset \\ c & otherwise \end{cases} \tag{5}$$

The quantities $P_{in}[B_i]$ and $P_{out}[B_i]$ are computed for each basic block B_i. Initially, $P_{in}[B_i] = \emptyset$ and $P_{out}[B_i] = gen[B_i]$. The set $gen[B_i]$ is defined by the set of cache blocks that are loaded during the execution of basic block B_i by predictable and unpredictable memory accesses. For unpredictable memory accesses we assume that all cache blocks are accessed. We define P for the entire cache by considering each cache set separately. The $\odot$ operator replaces a cache block if $gen[B_i]$ is not empty, otherwise the cache block from some incoming path $P_{in}[B_i]$ is taken. We set $P[B_i] = P_{out}[B_i]$ when the fixed point is reached which contains all persistent cache blocks at B_i.

The persistence analysis has to be performed for each unpredictable data access d. The *gen* set for the node containing d is constructed by all possible addresses. During the data flow analysis, d is ignored in all other nodes. Otherwise, some memory addresses would be loaded to the cache and the persistence analysis would not deliver the worst case.

4.3 Predictable data accesses

Global data flow analysis is used to calculate the data cache behavior for predictable memory accesses. The iterative data flow analysis, as described in section 4.2, can be applied with the following modifications: The $gen[B_i]$ set is defined by the predictable data accesses only. The replaced cache blocks by unpredictable array accesses can be ignored because their potential interference is accounted by the miss counter $C_m(B_i)$. At the end of this analysis step, the number of cache hits $B_i(hits)$ and misses $B_i(misses)$ for each node B_i are computed.

5 Timing analysis

After the cache access behavior for each node has been computed, we present now the last step of the analysis

framework: the global timing analysis. Integer linear programming is used to calculate the worst case data cache behavior (section 5.1) for the whole control flow graph. Unpredictable data accesses are then accounted by applying the pigeon-hole principle (section 5.2. In section 5.3 we summarize the assumptions and limitations of the whole analysis framework.

5.1 Integer linear programming

Integer linear programming (ILP) is an established method to find the worst case execution path in timing analysis [14] [24]. Based on the control flow graph of a program, a linear optimization problem is constructed that maximizes the flow through the program. Each node B_i contributes a constant execution time c_i and a variable x_i denotes the execution count of node B_i. In the following we focus on the contribution of data cache behavior and neglect instruction cache behavior. Therefore, our approach can be combined with any existing static timing analysis approach, e.g. that already considers instruction caches, pipelining and branch behavior. The objective function of the ILP is defined as:

$$max : \sum_{i \in B} c_i \cdot x_i \tag{6}$$

Where B denotes the set of all nodes of the control flow graph. A set of structural constraints is set up to model that the incoming flow of each node is equal to the outgoing flow:

$$\sum_{p \in pred(B_i)} e_{p,i} = x_i = \sum_{s \in succ(B_i)} e_{s,i} \tag{7}$$

where $e_{i,j}$ denotes an edge from node B_i to node B_j. The set of all predecessors is abbreviated as $pred(B_i)$, and the set of successors as $succ(B_i)$ for a node B_i. Additional constraints, e.g. for maximum number of loop iterations, can be defined by the engineer. The cache access time c_i for each basic block B_i is set to

$$c_i = c^{hit} \cdot B_i(hits) + c^{miss} \cdot B_i(misses) \tag{8}$$

where $B_i(hits)$ and $B_i(misses)$ denote the worst case number of cache hits and misses for basic block B_i of predictable data accesses, as computed in section 4.3. Cache hit and miss penalty time is denoted by c^{hit} and c^{miss} respectively. For unpredictable data accesses we apply the pigeon-hole principle.

5.2 Pigeon-hole principle

We extend the ILP to account for unpredictable memory accesses with the pigeon-hole principle [8], as described in s:dc:relatedwork. For a node B_i with a data dependent array d_k the execution count is separated into hits and misses. $Data(B_i)$ denotes the set of unpredictable data accesses.

$$x_i = x_{ik}^{hit} + x_{ik}^{miss} \quad \forall d_k \in Data(B_i) \tag{9}$$

For every *persistent* data access, which were computed in section 4.2 an additional in-equation regarding the pigeon-hole principle is added to the ILP:

$$x_{ik}^{miss} \leq range(d_k) \tag{10}$$

This in-equation reflects the fact that the number of misses is bounded by the maximum number of elements of the array since the array is persistent in cache. Then, the objective function is modified to

$$max : \sum_{i \in B} \left(\sum_{d_k \in Data(i)} \left(c^{hit} \cdot x_{ik}^{hit} + c^{miss} \cdot x_{ik}^{miss} \right) \right) + c^{miss} \cdot C_m(B_i) \cdot x_i + c_i \cdot x_i \tag{11}$$

The second sum represent the cache behavior for unpredictable memory accesses (pigeon-hole principle) together with equation 9and 10. The additional cache misses regarding existing cache contents is modeled by term $c^{miss} \cdot C_m(B_i) \cdot x_i$ which multiplies the cache miss penalty c^{miss} with the cache miss counter $C_m(B_i)$ and the execution count x_i of the basic block B_i. The last summand $c_i \cdot x_i$ is the cache behavior for predictable memory accesses (equation 8).

Figure 7 shows an example ILP for the CFG of figure 4. We assume that the array b has 10 elements. Note, that the loop statement B_5 is a SFP, therefore, no loop bound is necessary.

max: $c_1x_1 + c_2x_2 + c^{hit}x_{31}^{hit} + c^{miss}x_{31}^{miss} + c^{miss}C_m(B_3) + c_4x_4 + c_5x_5$

subject to

$x_1 = 1; x_1 = e_{1,2} + e_{1,3};$	// structural
$x_2 = e_{1,2} = e_{2,4}; x_3 = e_{1,3} = e_{3,4};$	// structural
$x_4 = e_{2,4} + e_{3,4} = e_{4,5}; x_5 = e_{4,5}$	// structural
$x_3 = x_{31}^{hit} + x_{31}^{miss}$	// pigeon-hole
$x_{31}^{miss} <= 10$	// pigeon-hole

Figure 7. Example ILP formulation.

5.3 Assumptions of analysis framework

Our approach assumes a direct mapped data cache (write through, no-write allocate), with a constant cache hit and cache miss penalty and an in-order execution model of the processor. Further, we disable optimizations beyond basic block boundaries. We do not consider pointer arithmetics and dynamic data structures on the heap. For memory trace simulation, test patterns for full branch coverage have to be available. For the experiment, we supplied the test data

manually. Branch coverage is often required for functional verification and should therefore be available for timing verification phase. The analysis framework does not support multiple function calls at the moment.

6 Experiments

This section presents results of the analysis framework for direct mapped caches.

Table 2. Benchmark description.

Task	Code	Data	C-Ln	Nodes	Name
τ_1	180	404	15	8	example1
τ_2	180	44	15	8	example2
τ_3	140	80	31	18	exchangesort
τ_4	1852	256	181	89	fft
τ_5	240	80	76	21	FIRFilter

Table 2 describes the benchmarks with total instruction memory (code) and data memory (data) in Bytes, number of c-lines (C-Ln) and number nodes (Nodes) of the control flow graph. The benchmark τ_1 is the example of figure 2. Benchmark τ_2 is based on τ_1 but the data access in the loop is input dependent and the size of the array is 10 elements. The maximum number of loop iterations is 1000. The benchmark τ_3 is a public domain sorting algorithm and τ_4, τ_5 are taken from [10]. We use g++ v3.0.4 with the option `-ast-original` to generate the abstract syntax tree, which is the input of data dependency analysis. We assume a 10 cycle cache miss penalty, 1 cycle cache hit time, and a fixed instruction length of 32 bits. In all experiments we use a direct mapped cache. The memory access trace was computed with the ARM9 processor simulator from ARM RealView Developer Suite, which contains the instruction as well as data addresses. Therefore, we simulated a unified cache. It could have also been applied to a separate data cache by filtering the data memory addresses. In the following we denote as the worst case execution time (WCET) the time to access this unified cache. The core execution time of the processor is not considered.

Table 3. WCET for data cache in [10^3clk].

Task	t_{cons}	t_{ana}	t_{sim}	$\frac{t_{cons}-t_{sim}}{t_{sim}}$	$\frac{t_{ana}-t_{sim}}{t_{sim}}$
τ_1	18.2	5.40	4.3	320%	25%
τ_2	78.3	29.3	20.1	290%	46%
τ_3	27.5	6.89	5.38	410%	28%
τ_4	807	285	205	290%	39%
τ_5	20.6	6.08	5.60	270%	8.6%

Results for the data cache timing behavior of these benchmarks for a 512 Byte direct mapped cache with block size of 16 bytes are shown in table 3 for a conservative analysis approach t_{cons}, for the proposed analysis framework t_{ana} and for cache simulation t_{sim}. The total worst case execution time (WCET) is given in 10^3 clock cycles (clk). The conservative estimate t_{cons} assumes two cache misses for each unpredictable memory access. The last two column expresses the overestimation of the conservative approach and our approach compared to cache simulation. The analysis precision of our approach is an improvement of an order of magnitude compared to the conservative approach.

Table 4. WCET in [10^3clk] for different cache sizes.

Cache, Task	t_{cons}	t_{ana}	t_{sim}	$\frac{t_{ana}-t_{sim}}{t_{sim}}$
$128-16\ \tau_2$	78.3	38.3	38.2	0.2%
$512-16\ \tau_2$	78.3	29.2	20.1	45%
$2048-16\ \tau_2$	78.3	29.3	20.1	46%
$128-16\ \tau_3$	32.3	11.7	11.1	5.4%
$512-16\ \tau_3$	27.5	6.89	5.38	28%
$2048-16\ \tau_3$	27.5	6.89	5.38	28%
$128-16\ \tau_5$	20.6	10.3	9.25	11%
$512-16\ \tau_5$	20.6	6.08	5.60	8.6%
$2048-16\ \tau_5$	20.6	3.77	3.54	6.5%

Table 4 shows the WCET for different cache sizes with 16 Byte cache block size only for some tasks due to space restrictions. While the conservative approach is very pessimistic, the proposed approach can be very close to the simulated results.

Table 5. WCET in [10^3clk] for different cache block sizes.

Cache, Task	t_{cons}	t_{ana}	t_{sim}	$\frac{t_{ana}-t_{sim}}{t_{sim}}$
$512-8\ \tau_2$	78.3	29.3	20.1	46%
$512-16\ \tau_2$	78.2	29.2	20.1	45%
$512-32\ \tau_2$	78.2	29.2	20.2	45%
$512-8\ \tau_3$	27.7	7.13	5.49	30%
$512-16\ \tau_3$	27.5	6.89	5.38	28%
$512-32\ \tau_3$	27.2	6.67	5.38	24%
$512-8\ \tau_5$	22.7	6.47	5.89	9.8%
$512-16\ \tau_5$	20.6	6.08	5.60	8.6%
$512-32\ \tau_5$	20.1	6.39	5.82	9.8%

Table 5 shows the results for different cache block sizes for a 512 Byte cache which shows that the deviation for different cache block sizes is small.

In summary, the our results are improved by an order of magnitude compared to conservative analysis. Compared to cache simulation our approach yields an overestimation of

8% to 46%. Note, that simulation results might not include the real worst case behavior. Therefore analysis results cannot directly be compared to simulation results. A second reason is that *nearly* input independent data access behavior cannot be detected by our analysis. E.g. memory accesses that are not fully input independent but contain some regular access patterns.

The time-complexity of the analysis is very low. For each task, the entire analysis including data dependency classification, memory mapping, data cache analysis, construction and solution of the ILP took less than one minute for calculating all nine cache configurations.

7 Conclusion

In this paper we have proposed a novel worst case timing analysis framework for data caches. Input data dependency has been addressed as the key issue to deliver tight timing bounds. We have presented an analysis for unpredictable data accesses as well as predictable data accesses. The analysis framework is a combination of data-flow analysis, pigeon-hole principle and integer linear programming. First experiments are very promising and we wish to extend the framework for more complex programs with function calls and associative data caches.

8 Acknowledgments

We would like to thank Isabelle Puaut, Jean-Francois Deverge, Monica Rosen and the anonymous reviewers for their helpful comments.

References

[1] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, GB, 1988.

[2] F. Bodin and I. Puaut. A WCET-oriented static branch prediction scheme for real-time systems. In *ECRTS*, Palma de Mallorca, Spain, July 2005.

[3] M. Campoy, A. P. Ivars, and J. V. Busquets-Mataix. Static use of locking caches in multitask preemptive real-time systems. In *IEEE Real-Time Embedded System Workshop*, December 2001.

[4] A. Datta, S. Choudhury, A. Basu, H. Tomiyama, and N. Dutt. Satisfying timing constraints of preemptive real-time tasks through task layout technique. In *IEEE VLSI Design*, pages 97–102, January 2001.

[5] C. Ferdinand and R. Wilhelm. On predicting data cache behaviour for real-time systems. In *ACM SIGPLAN Workshop 1998 on Languages, Compilers, and Tools for Embedded System*, 1998.

[6] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[7] J. Gustafsson, B. Lisper, P. Puschner, and R. Kirner. Input-dependency analysis for hard real-time software. In *Workshop on Object-oriented Real-time Dependable Systems*, Capri Island, Italy, 2003.

[8] S.-K. Kim, S. L. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 1996.

[9] D. B. Kirk. SMART(strategic memory allocation for real-tim) cache design. In *Real-Time Systems Symposium*, pages 229–239, Santa Monica, CA, USA, Dec. 1989. IEEE Computer Society Press.

[10] C.-G. Lee, J. Hahn, and Y.-M. S. et al. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on computers*, 47(6):700–713, June 1998.

[11] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*, pages 254–263, 1996.

[12] J. Liedtke, H. Härtig, and M. Hohmuth. Os-controlled cache predictability for real-time systems. In *RTAS*, Montreal, Canada, June 9-11 1997.

[13] T. Lundqvist and P. Stenström. A method to improve the estimated worst-case performance of data caching. In *Intl. Conference on Real-Time Computing Systems and Applications (RTCSA)*, pages 255–262, 1999.

[14] S. Malik and Y.-T. S. Li. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic Publishers, 1999.

[15] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *CODES+ISSS'03*, Newport Beach, CA, USA, Oct. 2003.

[16] P. R. Panda, N. D. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-On-Chip: Optimizations and Exploration*. Kluwer Academic Publishers,Norwell, MA, 1999.

[17] I. Puaut and D. Decotigny. Low-complexity algorihtms for static cache locking in multitasking hard real-time systems. In *RTSS*, 2002.

[18] H. Ramaprasad and F. Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 148–157, 2005.

[19] J. Schneider and C. Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. *SIGPLAN Not. LCTES'99*, 34(7):35–44, 1999.

[20] J. Staschulat and R. Ernst. Multiple process execution in cache related preemption delay analysis. In *EMSOFT*, Pisa, Italy, Sept. 2004.

[21] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. In *IEEE Real-Time Systems Symposium*, 2003.

[22] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. Timing analysis for data and wrap-around fill caches. *Real-Time Systems*, 17(2-3):209–233, 1999.

[23] F. Wolf, R. Ernst, and W. Ye. Path clustering in software timing analysis. *IEEE Transactions on VLSI Systems*, 9(6):773–782, 2001.

[24] F. Wolf, J. Staschulat, and R. Ernst. Hybrid cache analysis in running time verification of embedded software. *Journal of Design Automation for Embedded Systems*, 7(3):271–295, 2002.