Bounding Worst-Case Data Cache Performance by Using Stack Distance

Yu Liu and Wei Zhang
Department of Electrical and Computer Engineering
Southern Illinois University Carbondale
Carbondale, IL 62901
{liu,zhang}@engr.siu.edu

Received 4 May 2009; Revised 28 October 2009; Accepted 4 November 2009

Worst-case execution time (WCET) analysis is critical for hard real-time systems to ensure that different tasks can meet their respective deadlines. While significant progress has been made for WCET analysis of instruction caches, the data cache timing analysis, especially for set-associative data caches, is rather limited. This paper proposes an approach to safely and tightly bounding data cache performance by computing the worst-case stack distance of data cache accesses. Our approach can not only be applied to direct-mapped caches, but also be used for set-associative or even fully-associative caches without increasing the complexity of analysis. Moreover, the proposed approach can statically categorize worst-case data cache misses into cold, conflict, and capacity misses, which can provide useful insights for designers to enhance the worst-case data cache performance. Our evaluation shows that the proposed data cache timing analysis technique can safely and accurately estimate the worst-case data cache performance, and the overestimation as compared to the observed worst-case data cache misses is within 1% on average.

Categories and Subject Descriptors: C3 [**SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS**] Real-time and Embedded Systems; J7 [**COMPUTERS IN OTHER SYSTEMS**] Real-time

General Terms: Performance, Reliability

Additional Key Words and Phrases: Static Timing Analysis, Worst-case Execution Time, Stack Distance, Data Caches

1. INTRODUCTION

Real-time systems ranging from aircraft controllers to pace makers have been widely used in our society. In those systems, it is important to know the worst-case execution time (WCET) of each real-time task to ensure that different tasks can meet their

respective deadlines. While the actual execution time of a given task can be measured on a particular platform, the WCET estimate based on measurement alone is unsafe because it is not feasible to exhaust all the possible program paths, especially for applications with complex control flows. Consequently, the static analysis technique (i.e., WCET analysis) has become a promising approach to obtaining the worst-case execution time for real-time tasks.

The WCET of a real-time task, however, is not only dependent on the inputs and program behavior, but also determined by the timing information of the underlying processor. Unfortunately, modern microprocessors are generally designed to improve the average-case performance, and some architectural features such as cache memories, branch prediction and speculative execution are detrimental to the worst-case execution time and can substantially complicate the WCET analysis [Berg et al. 2004; Rochange and Sainrat 2002]. In the last two decades, there has been much work on WCET analysis for microprocessors with advanced architectural features [Wilhelm et. al 2007]. While significant progress has been made in static instruction cache analysis [Arnold et al. 1994; Healy et al. 1995; Li et al. 1995; Lim et al. 1994]; relatively, improvement in data cache WCET analysis is rather limited. The main reason is that unlike instruction accesses, memory addresses of data accesses can change at runtime. For example, a reference to an array element in a loop is likely to access different data with varied memory addresses in different loop iterations (although the instruction of this data access is fixed across loop iterations). In addition, data caches are often set-associative so as to reduce the conflict data misses (while instruction caches could be direct-mapped to minimize access latency). As a result, timing analysis for (set-associate) data caches can be very complex, due to the need to estimate the worst-case data access history for each cache set with multiple ways.

This paper propose an approach to tightly bounding the worst-case data cache performance by computing stack distance [Beyls and D'Hollander 2001; Cascaval and Padua 2003] statically across different program paths. Compared with prior work [White et al. 1997; Li et al. 1996; Ferdinand and Wilhelm 1998; Ramaprasad and Mueller 2005; Staschulate and Ernst 2006] on static data cache analysis, this paper makes the following major contributions:

- The proposed approach can be easily applied to not only direct-mapped caches, but also set-associative and fully-associative caches, without increasing the analysis complexity.
- Our approach can statically categorize worst-case data cache misses into cold misses, conflict misses and capacity misses, which can provide useful insights to cache and software designers to enhance the worst-case data cache performance. To the best of our knowledge, no existing data cache WCET analysis techniques [White et al. 1997; Li et al. 1996; Ferdinand and Wihelm 1998; Ramaprasad and Mueller 2005; Staschulate and Ernst 2006] can provide such specific information of worst-case data cache misses.
- We extensively evaluate the proposed static timing analysis approach on a diverse set of benchmarks for a variety of data caches. The evaluation shows that the

proposed WCET analysis can safely and very accurately estimate the worst-case data cache performance, and the overestimation as compared to the observed worst-case data cache misses through simulation is within 1% on average.

The rest of this paper is organized as follows. Section 2 introduces the worst-case data cache analysis approach based on computing stack distance. Section 3 describes the evaluation methodology and Section 4 gives the experimental results. The related work is discussed in Section 5. Finally, we draw conclusions in Section 6.

2. STACK DISTANCE BASED WORST-CASE DATA CACHE ANALYSIS

Stack distance [Beyls and Hollander 2001; Cascaval and Padua 2003] has been widely used in cache performance analysis; however, prior work on stack distance has concentrated on studying the average-case cache behavior, which can be quite different from the worst-case behavior. To the best of our knowledge, this is the first work to analyze the worst-case data cache behavior by using stack distance.

Stack distance of a memory access can be defined as the number of accesses to unique addresses made since the last reference to the requested data [Beyls and Hollander 2001]. The stack distance has an interesting property: *in a fully-associative LRU cache with n lines, a reference with stack distance $d < n$ will hit, and a reference with stack distance $d \geq n$ will miss.* (the detailed proof can be found at [Beyls and Hollander 2001]). Therefore, cache misses can be easily classified into cold, conflict or capacity misses based on the stack distance. More specifically, suppose a d-way set-associative cache has N lines, and the stack distance of a cache access a is s, then:

- a is a hit, if $s < d$.
- a is a conflict miss, if $d \leq s < N$.
- a is a capacity miss, if $N \leq s < \infty$
- a is a cold miss, if $s = \infty$ (i.e., a has not been accessed before).

To analyze the worst-case data cache behavior by using the stack distance, it is a necessity to generate data cache access traces for different program paths and then calculate the worst-case data cache misses based on the worst-case data cache access trace (instead of an average-case trace). The framework of our data cache timing analysis technique is shown in Figure 1. As we can see, the *C* source file is firstly compiled by the Trimaran compiler [Trimaran 2009]. According to the data declaration and relative address information from the compiled code, a data address calculator generates virtual addresses for data cache accesses on different program paths. The stack distance calculator then calculates the stack distance for each data cache access, built upon which the cache hits or misses can be derived. Finally, the timing analyzer enumerates data cache accesses on different program paths to compute the worst-case cache performance, as well as the WCET.

In the rest of the section, we will explain static data address generation in subsection 2.1. The stack distance computation will be introduced in subsection 2.2, and the worst-case data cache performance calculation will be presented in subsection 2.3.

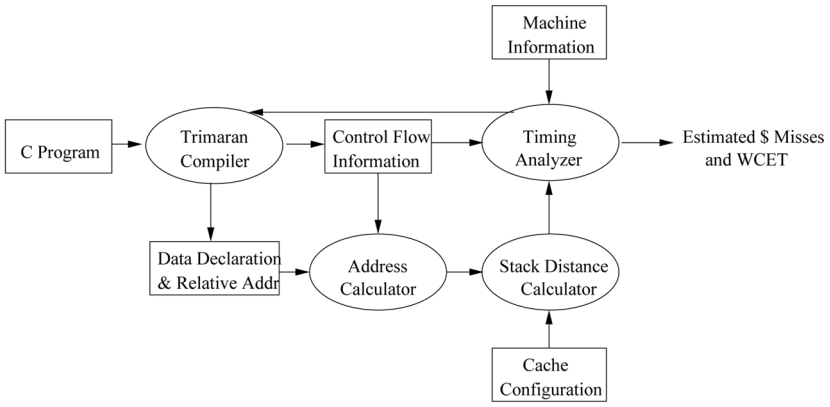


Figure 1. Framework of Stack Distance Based Data Cache Timing Analysis Tool.

2.1 Static Data Address Generation

Our static data address generation is based on prior work in [White et al. 1997], which consists of two steps, i.e., relative address calculation and virtual address calculation. Before introducing the algorithms to generate data addresses, we would like to point out the constrains of our approach.

In this work, we have made the following three assumptions. First, there is no dynamic memory allocation (i.e., memory reference in heap) in the source program. Second, the maximal number of loop iterations can be either statically analyzed or annotated by users. Third, we assume that the page size of the virtual memory system is an integer multiple of the data cache size, which is not uncommon. For example, the MicroSPARC I has a 4KB page size and a 2KB data cache [White et al. 1997]. Therefore, we can use part of the virtual address, specifically the page offset, to index the cache. In other words, both a virtual address and its corresponding physical address will be mapped to the same line in the data cache.

Before the virtual address generation, the timing analysis tool first calculates the relative address for each load and store instruction. There are three scenarios for the relative address calculation:

- Static data references (i.e., global scalars: their relative addresses can be directly calculated through the data declaration information available from the compiler. Typically, data references of this kind occur outside loops, thus their relative addresses are zero.
- Scalar references to the run-time stack (i.e., local scalars): Normally, the compiler will firstly try to allocate local scalars to the registers. If all available registers are used (i.e., register spills), the remaining local scalars will be allocated to the memory due to insufficient registers. For scalars spilled to memory, our algorithm can capture their addresses in the *load/store* instructions, which are statically available. Their relative addresses can be calculated as a set of addresses based on the sequence of calls that are associated with an invocation of a function, which can be determined by statically analyzing the call graph. In addition, for these

scalars allocated to the registers, we do not need to take care of them for the data cache analysis, since they do not need to access the cache.

– Calculated address references (i.e., arrays and pointers): these data references normally occur inside loops, which can be referred by two different methods in C language, i.e., by array indexes or pointers. We will introduce two algorithms (see Section 2.1.1 and 2.1.2) to calculate the relative addresses for references by arrays and pointers, respectively. All of our static analysis focuses on the assembly codes, and the address generation patterns for the array-based and pointer-based data references are not exact the same in the assembly codes generated by Trimaran compiler. Thus, we need to analyze them separately to calculate the relative addresses. The proposed analysis needs to statically compute the memory address of each *load/store* instruction that needs to access data cache. Thus our analysis depends on the code generator in terms of the format of instruction, the addressing mode to generate the memory addresses. Although our analysis in this paper focuses on the compiling result of Trimaran, we believe our approach can be easily adapted to other compilers as well.

To statically calculate addresses for both arrays and pointers, the address generator needs to determine, for each loop, the set of the primary induction variables, their initial values (e.g *init1*, *init2*, ..., and *initn* in the following algorithms), strides (e.g. *stride1*, *stride2*, ..., and *striden* in the following algorithms) and the number of iterations for each loop (e.g. *time1*, *time2*, ..., and *timen* in the following algorithms).

In this paper, we refer the chain of *addition/subtraction* operations involved in relative address generation as intermediate *additions/subtractions*. Particularly, the intermediate *addition/subtraction* that stores the base address information in the source operand is called the last intermediate *addition/subtraction* (i.e., the intermediate *addition/subtraction n*). The loop corresponding to the relative address generation for

Algorithm: Relative Address Generation for Memory References through Array Indexs

```
1:  scan the assembly codes to get all load/store operatioins in the loop;
2:
3:  /*iterate all load/store op in the loop*/
4:  while (there is a load/store op in the loop)
5:  {
6:    scan assembly codes to find all intermediate arithmetic ops;
7:    for (i=1 to n) //n is the total number of intermidate arithmetic ops.
8:    {
9:      analyze intermediate arithmetic op i to get info on seed register, and stride value;
10:     analyze the seed register to find seed loop info, e.g. loop exection time & initial value;
11:   }
12:   get the base address info from the last intermediate arithmetic op;
13:
14:   /*generate all relative addresses for this load/store ops*/
15:   for (seed1=init1 to time1)
16:     for (seed2=init2 to time2)
17:       ... ...
18:         for (seedn=initn to timen)
19:           relative address=seed1*stride1+seed2*stride2+...+seedn*striden;
20: }
```

Figure 2. The Algorithm to Calculate Addresses for Array References.

```
/* the data type of array a is integer */
for (int i = 0; i<5; i++)
  for (int j=0; j<5; j++)
    a[i][j] = 0;
```

(Source Code)

```
bb2 (
weight(1)
op 5 (MOVE [br<1:i gpr 4>] [i<0>]
)
bb3 (
weight(5)
op 7 (MOVE [br<2:i gpr 3>] [i<0>]
)
bb4(
weight(25)
op 9 (MPY_W [br<4:i gpr 2>] [br<1:i gpr 4> i<20>]
op 11 (MPY_W [br<6:i gpr 5>] [br<2:i gpr 3> i<4>]
op 14 (ADD_W [br<2:i gpr 3>] [br<2:i gpr 3> i<1>]
op 10 (ADDL_W [br<5:i gpr 6>] [l:g_abs<_a> br<4:i gpr 2>]
op 6 (ADD_W [br<34:i gpr 7>] [br<5:i gpr 6> br<6:i gpr 5>]
op 12 (S_W_C1 [] [br<34:i gpr 7> i<0>]
)
bb5 (
weight(5)
op 17 (ADD_W [br<1:i gpr 4>] [br<1:i gpr 4> i<1>]
}
```

(Trimaran Rebel Code)

Figure 3. An Example of Data Address Calculation for Array References.

the *load/store* operation is called the *seed loop* in this paper, and the primary induction register of the *seed loop* is referred as the *seed register*. The stride value for each *seed loop* can be found in a *multiply* operation, in which the *1st* source operand is the *seed register* and the *2nd* source operand contains the stride value.

2.1.1 Address calculation for array references

Algorithm: Relative Address Generation for Memeory References through Pointers

```
1:  scan the assembly codes to get all load/store operations in the loop;
2:
3:  /*iterate all load/store operations in the loop*/
4:  while (there is load/store op in loops)
5:  {
6:    compute base address through intermediate mov ops;
7:    compute the stride value through the intermediate add/sub ops;
8:    use seed register to get info of the seed loop, e.g. loop execution time, and initial value;
9:
10:   /*generate all relative addresses for this load/store op*/
11:   for (seed=init to time)
12:     relative address=seed*stride;
13: }
```

Figure 4. The Algorithm to Calculate Addresses for Pointer References.

The algorithm to statically calculate addresses for array references is given in Figure 2. The basic idea of this algorithm is to perform data flow analysis on the compiled code to get the information about address changing strides, initial values, and the number of iterations for each loop, based on which the static address generator can calculate the sequence of relative addresses for array references. First, we get all *load/store* operations from the Rebel code. Then, we get the information needed for relative address generation through scanning all *addition*, *multiply*, and *move* operations respectively. Last, we use all the information gathered to generate the relative addresses following the array-based address generation mode.

To illustrate this algorithm, an example is given in Figure 3. In this example, there is only a *store* operation (i.e., *op* 12) that accesses memory. In order to get all the relative addresses of this *store* instruction in all the loop iterations, the static data address generator takes the following steps:

(1) Scan all *load/store* operations in Rebel code to find the *store* operation, i.e., *op 12* (line 1). The SRC1 register of this operation contains the memory address, and the number of this register is *r7*.
(2) Scan all *addition* operations on the data dependence chain to find the intermediate *addition* (i.e., *op 6*) whose DEST1 register is *r7* (line 4-6). The SRC1 register and SRC2 register of *op 6* are *r6* and *r5*, respectively.
(3) Scan all *multiply* operations (line 7-11) (i.e., *op 11* in this example) whose DEST1 register is the second address source register (i.e., *r5*) in order to find the *multiply* operation for the primary induction register of the inner loop. As a result, *r3* is identified as the primary induction register of the inner loop, and the stride value of the outer loop is *4*, as specified in SRC2 of this operation.
(4) Scan all *multiply* operations (line 7-11) (i.e., *op 9* in this example) whose DEST1 register is the intermediate register (i.e., *r2*) for primary induction register of the outer loop to find the *multiply* operation for the primary induction register of the outer loop. As a result, *r4* is identified as the primary induction register of the outer loop, and the stride value of the outer loop is *20*, as specified in SRC2 of this operation.
(5) Scan all *move* operations (line 7-11) (i.e., *op 5* and *op 7* in this example) whose DEST1 registers are primary induction registers. These *move* operations hold the initial values of inner and outer loops, which are 0 for this example.
(6) Scan all intermediate *addition* operations (line 12) whose DEST1 register is the first address source register (i.e., *r6*) and the result is *op 10*, which adds the base address with the relative address. In *op 10*, the label of the base address is *_a*, and *r2* specifies the basic induction register that controls the loop execution (line 12).
(7) Then by applying the algorithm of relative address calculation (line 14-19), the following relative addresses will be generated for the store operation (i.e., *12*) of this code segment: *0*, *4*, *8*, *12*, *16*, *20*, *24*, *28*, *32*, *36*, *40*, *44*, *48*, *52*, *56*, *60*, *64*, *68*, *72*, *76*, *80*, *84*, *88*, *92*, *96*.

2.1.2 Address calculation for pointer references

Another way to access data in memory is to use pointers. Once the value of a pointer is modified, the memory address to which this pointer points to will also be changed.

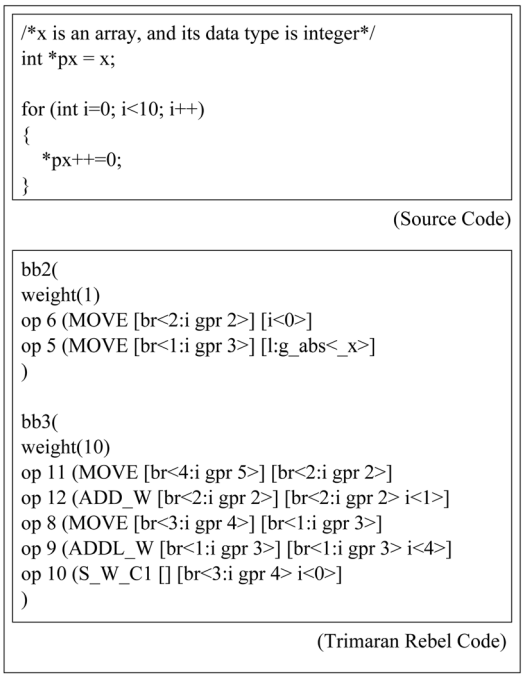


Figure 5. An Example of Calculating Relative Addresses for Pointer References.

Therefore, the static address generator needs to obtain the information on the stride of the address changes by this pointer, as well as the information on the loop where this pointer is located, based on which it can statically generate the sequence of relative addresses for pointer-based memory references. The algorithm to calculate relative addresses for pointer references is described in Figure 4. First, we get all *load/store* operations from the Rebel code. Then, we get the information needed for relative address generation through scanning all *addition* and *move* operations respectively. Last, we use all the information gathered to generate the relative addresses following the pointer-based address generation mode.

To illustrate this algorithm, an example is given in Figure 5. In this example, we need to calculate relative addresses for C code **px++=0* in the loop. In order to get all relative addresses for this *store* instruction, the static address generator takes the following steps:

(1) Scan all *load/store* operations in the Rebel code (line 1). Consequently, the \it store operation (*op 10*) is found, whose SRC1 register (i.e., *r4*) contains the memory address.
(2) Scan all *move* operations (line 4-6) to identify one (i.e., *op 8*) whose DEST1 register is *r4*, and whose SRC1 register (*r3*) of this operation contains the memory address.
(3) Scan all *move* operations (line 4-6) to find the one (i.e., *op 5*) whose DEST1 register is *r3*. From this operation (i.e., *op 5*), the algorithm can identify the label of base address, which is *_x* in this example.

Algorithms 3: Virtual Address Trace Generation

```
1:  while (there are load/store operations in program)
2:     while (there are relative addresses for this load/store op)
3:     {
4:        add base address of this op with its relative address as virtual address;
5:        save this virtual address for this op into address trace of this op;
6:     }
7:  while (there is loop in program)
8:  {
9:    get the nesting level information for this loop;
10:   while (i<the number of iterations of this loop)
11:   {
12:      fetch virtual address of each load/store operations in this loop;
13:      merge these fetched virtual addresses according to their positions in loop;
14:      i++;
15:   }
16: }
17:  while (there are loop groups formed by loop nesting in program)
18:    for (from innermost loop to outermost loop in this loop group)
19:    {
20:      if (this loop is innermost loop)
21:        just copy its address trace into adddress trace of this loop group;
22:      else
23:        merge address trace of this loop with address trace of its nested loop;
24:      save these virtual addresses for this loop group into its address trace;
25:    }
26: merge the address traces of the nested loops with virtual addresses outside loops;
```

Figure 6. The Algorithm to Calculate Virtual Address for Each Data Reference and Merge the Address Trace.

(4) Scan all *addition* operations (line 7) whose DEST1 register is *r3*, and SRC2 of this addition (i.e., *op 9*) contains the stride value, which is 4 for this example.
(5) Scan all operations to find the primary induction register of this loop (line 8), and get the information of the number of loop iterations (i.e., 10) and the initial value (i.e., 0) for this loop.
(6) Then based on the relative address calculation process (line 11-13), a sequence of relative addresses are generated as the following: *0*, *4*, *8*, *12*, *16*, *20*, *24*, *28*, *32*, *36*.

2.1.3 Virtual Address and Address Trace Generation

After getting relative addresses, we need to generate the virtual address for each *load/store* operation in the program, including these inside the loops and outside the loops. The address calculator generates the virtual address by adding the base address with the relative address. Also, we need to merge these virtual addresses following their sequence on a specific path to get the virtual address trace on this path. Later, the address traces of different paths inside and outside the loops will be processed by the stack distance to predict the cache behavior. Comparing these processing results from the stack distance, we can determine the worst-case cache behavior, which will be detailed in subsection 2.3.

More specifically, first the loops should be processed in a bottom-up manner to generate its virtual address trace due to the loop nesting. It means that we process

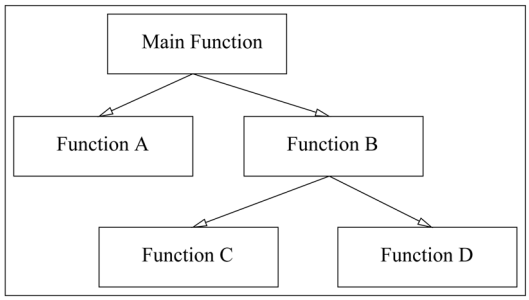


Figure 7. An Example of Generating the Address Trace for the Program with Function Calls.

nested loops from the innermost one to the outermost one. In another word, the virtual address trace of the loop is generated until all of its child loops are processed. Second, we merge the virtual address traces inside loops with these outside loops following their sequence on a specific path to get the whole virtual address traces.

A detailed algorithm is summarized in Figure 6. In this algorithm, we first generate the virtual addresses for each *load/store* operation (Line 1-6). Second, we generate the address trace inside each loop in the program (Line 7-16). Third, we merge the address trace inside the nested loops following the bottom-up manner mentioned above (Line 17-25). Last, we merge the address traces inside the nested loops with addresses outside the loops to generate the whole address trace (Line 26).

In our work, all functions in the benchmarks are inlined in our work. Thus, we do not have function call handling. However, we believe that our method can be easily extended to handle the function calls. Thus, we propose the following method to generate the virtual memory address trace for the program with function calls following the tree style call graph. When we generate the memory address trace for the whole program, we need to generate the address traces for functions without any other function calls (i.e., leaf functions) at first, and then process their caller functions following the bottom-up manner till the root of call graph tree. Obviously, the main function (i.e., the root function) is the last one to be processed, and its memory address trace is the final one for the whole program. Also, we need to compare the data cache performance of different paths in each function by processing their address traces through the stack distance. Then, we only merge the address trace of the worst-case data cache performance path of this function to the address trace of its caller functions.

An example is given in Figure 7. First, we generate the address traces for the leaf functions separately, including the function *A*, *C* and *D*. Also, we compare the data cache performance of different paths through the stack distance in each of these functions to obtain the address trace of the worst-case path. Second, we merge the worst-case address traces of the function *C* and *D* into different paths of the function *B*. Following the same comparison, we can get the worst-case address trace for the function *B*. Last, we merge the address traces of the function *A* and *B* with different paths of the *main* function, do the comparison among the paths, and then get the worst-case address trace for the whole program.

Table I. An Example of Calculating Stack Distance Based on Address and Cache Line Granularity Respectively, for a Cache Line Size of 16 Bytes.

Reference No	1	2	3	4	5	6	7
Address Stack Distance (addr)	0 •	32 •	24 •	96 •	8 •	24 •	100 •
Cache Line Stack Distance (line)	0 •	2 •	1 •	6 •	0 3	1 2	6 2

2.2 Calculating Stack Distance

Once the virtual address is generated for each *load/store* instruction, the analysis tool can calculate the stack distance for each data cache reference. With the data cache configuration information (i.e., cache size, cache line size, associative information, etc.), data cache behavior can be statically predicted for all data references, based on the stack distance property introduced in the beginning of Section 2.

It should be noted that both temporal locality and spatial locality of data references need to be considered to calculate the stack distance accurately. More specifically, the timing analysis tool measures the stack distance with cache line granularity, not address granularity. The example given in Table I shows the difference of stack distance calculation between using address granularity and using cache line granularity, assuming the cache line size is 16 bytes.

Built upon the prior work in *stack3*, the stack distance calculator takes the following five steps to compute stack distance for each data cache reference:

– Input: read the next reference from the address trace file generated by the virtual address generator.
– Find: search the whole stack to find the cache line that this reference accesses and then determine its stack distance.
– Update: if the reference is the first access to that trace element, push this reference into the top of the stack. Otherwise, move the reference θ found in the stack to the top of the stack, and all the references between the top and the original θ location are pushed down one position. The references below the original θ location will not be affected.
– Calculation: update stack distance histogram, i.e., $S(\theta)$, according to the stack distance of this reference. More specifically, $S(\theta)$ can be calculated as follows: (1) if the reference is the first access to that element, set $\theta = \infty$, increment $S(\infty)$ and push this reference into the top of the stack; (2) if this data has been previously referenced in the stack, let θ be the distance from the top of the stack to the position at which the reference is found, increment $S(\theta)$, and move this reference to the top of the stack.
– Categorization: based on the stack distance of each data cache reference, we can categorize it into either cache hit or one of the cold, conflict or capacity misses, according to the stack distance property and its use in cache classification described in the beginning of Section 2.

Algorithm: Computing the Worst−Case Data Cache Misses for Loops

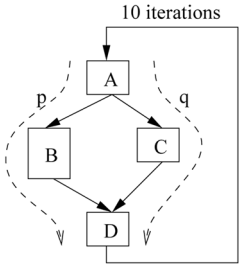
```
1:  for (each loop L in the program){
2:     if (no branch in L){
3:        generate data trace for a single path;
4:        calculate the data cache misses based on the stack distances;
5:     }
6:     else {
7:        for (each branch B in L)
8:           generate data traces on both paths of B;
9:        for (each generated data trace T){
10:          calculate total data cache misses for T based on stack distances;
11:          calculate the number of data cache misses D(T,I) of T for each iteration I;
12:       }
13:       for (i=1 to M) { // M is 2 to the power of the number of branches in L
14:          for (j = 1 to N) {// N is the number of iterations for Loop L
15:             sort D(i,j) in descreasing order
16:       worst−case−Dcache−misses (L) = sum of the first N elements in sorted D(i,j);
17:    }
18: }
```

Figure 8. The Algorithm to Calculate Worst-case Data Cache Misses for Loops.

2.3 Computing Worst-Case Data Cache Misses

So far we have focused on analyzing the data cache misses for *load/store* instructions on a given program path. Since there are many possible paths in a program, the timing analyzer needs to find the largest number of data cache misses to bound the worst-case data cache performance. For paths generated by the branches outside loops, since there are relatively a small number of different paths, especially for small real-time kernels, the timing analyzer can relatively more efficiently (as compared to enumerate all paths inside loops) exhaust all the data traces on different paths to estimate the worst-case data cache misses. Although we still need to analyze a large number of paths for the large benchmarks, compared to the number of paths generated by loops, the number of paths outside the loop is still limited. For instance, even for a loop with two inside paths and 100 iterations, it can generate 2^{100} number of paths. Therefore, paths outside loops are relatively easier to enumerate as compared to paths inside loops. Thus, for paths generated by branches inside loops, it is generally infeasible to exhaust all the different paths, which could be exponential to the number of loop iterations. In this case, we use a path-based static analysis algorithm as depicted in Figure 8 to effectively estimate the worst-case data cache misses.

It should be noted that our algorithm assumes that there is no conflict between data cache accesses on different paths (although data cache conflicts within a single path are allowed and can be analyzed by our approach). Also, the benchmarks used in our work are adapted to provide large number of data accesses and match this assumption. This assumption greatly reduces the number of paths that the timing analyzer needs to enumerate, which is also a reasonable assumption. This is because



(a)

p1	p2	p3	p4	p5	p6	p7	p8	p9	p10
8	2	2	2	2	2	2	2	2	2
q1	q2	q3	q4	q5	q6	q7	q8	q9	q10
4	1	1	1	1	1	1	1	1	1

(b)

10 largest number of cache misses: 8 4 2 2 2 2 2 2 2 2
Worst−case path: p1−q1−p2−p3−p4−p5−p6−p7−p8−p9
Worst−case Dcache misses:8+4+2+2+2+2+2+2+2+2=28

(c)

Figure 9. An Example of Computing Worst-case Data Cache Misses for Loops.

if data accesses on two different paths of a loop have conflicts, they may alternatively replace each other repeatedly in different loop iterations, thus leading to excessive cache misses for the program. In that case, typically compilers can perform data layout optimizations [Muchnick 1997] to eliminate these inter-path cache conflicts within loops.

As shown in Figure 8, the timing analyzer examines each loop in the program (line 1). If a loop has no branch, then there is only a single path, and the worst-case data cache timing analysis becomes trivial (line 2-5). For loops with branches, the algorithm generates data traces on both paths for each conditional branch (line 7-8). By calculating stack distances for data references on each path, the timing analyzer can estimate the number of data cache misses on each loop iteration for each data trace $D(T, i)$, by taking into account the cache associativity, size and block size (line 9-12). The timing analyzer then sorts all the $D(T, i)$ in decreasing order, and the sum of the first N (i.e., the number of loop iterations) $D(T, i)$ (i.e., the N largest $D(T, i)$) will be the worst-case number of data cache misses for this loop.

To illustrate this algorithm, an example is given in Figure 9. Figure 9(a) depicts the control flow graph of a loop with two paths (i.e., this loop contains only one conditional branch), including path p (A–B–D) and path q (A–C–D). Suppose the total number of loop iterations is 10. Based on the algorithm described in Figure 8, the timing analyzer computes the stack distance and derives the number of cache misses on each path for each loop iteration (i.e., p_i and q_i, $1 \le i \le 10$), which are given in Figure 9(b). Our algorithm then sorts all the p_i and q_i ($1 \le i \le 10$) in decreasing order, and finds the first 10 largest numbers, which are shown in Figure 9(c). The worst-case number of data cache misses is then calculated as the sum of the data cache misses on those 10 iterations, which is 28 for this example.

It should be noted that the algorithm given in Figure 8 can be computed efficiently. Suppose a loop has N iterations and B conditional branches, the complexity of our algorithm is $O(2^B * N)$. Since a loop typically only contains a small number of branches, the time efficiency of this algorithm is most likely linear. By comparison, the complexity of a naive algorithm to exhaust all the paths in a loop for deriving the worst-case data cache misses is $O(2^{B*N})$, which is exponential and prohibitively expensive to compute.

2.4 Analysis with Relaxed Limitation

In addition, we believe that our algorithm can be extended to cover the scenarios that there is conflict between data cache accesses on different paths. First, we still use our algorithm to find the number of worst-case data cache missed by assuming there is no inter-path conflict as the *base cache misses*. Then, we interleave different paths in the loop to find the largest number of data cache misses due to inter-path conflict as the *extra cache misses*. Last, we add up the *base cache misses* and the *extra cache misses* to get an upper bound of worst-case data cache misses. Although this method will bring some overestimation, it is surely a safe estimation for worst-case data cache performance estimation.

Let's still use the example in Figure 9 to illustrate our extended method. Now, we suppose that there is conflict between data cache accesses on the path p and q. First, we can still get 28 data cache misses as the *base cache misses* following our original method. Then, we compare the two path interleaving pattern pq and qp respectively. Let's suppose the number of data cache misses of the pattern pq due to inter-path conflict is 2, while it is 1 for the pattern qp. Thus, we choose 2 as the *extra cache misses*. Since the number of iteration of the loop is 10, we can repeat this interleaving pattern 5 times. Therefore, the total *extra cache misses* is 10 by 2 times 5. Last, we can get the upper bound of worst-case data cache misses by 28 plus 10, which is 38 in total. Clearly, there could not be "worse" data cache performance than this; thus this analysis can obtain safe yet maybe overestimated worst-case data cache misses.

3. EVALUATION METHODOLOGY

To evaluate the proposed worst-case data cache timing analysis approach, we compare the estimated worst-case data cache misses with the observed worst-case data cache misses through extensive simulation. We simulate a VLIW processor based on the HPL-PD architecture [Kathail et al. 2000] by using Trimaran compiler/simulator infrastructure [Trimaran 2009]. The data cache timing analyzer has been implemented as independent modules in the backend compiler Elcor. The important parameters of the baseline VLIW processor are given in Table II. It should be noted

Table II. Configuration Parameters and their Values in the base Configuration of the Simulated VLIW Processor.

Configuration Parameter	Value
Processor	
Functional Units	2 integer FUs 2 floating-point FUs 1 load/store unit 1 branch unit
Register File	16 global registers
Cache and Memory Hierarchy	
L1 Data Cache	8KB, 2-way, LRU, 32B blocks 1 cycle latency
L1 Instruction Cache	perfect
Memory	10/100 cycle, unlimited size

Table III. The Salient Characteristics of the Selected Benchmarks.

Benchmarks	Source	Exec Cycles	*D* Accesses	*D* Misses	*D* Miss Rate
dag	Trimaran	6431496	1200001	1247	0.104%
eight	Trimaran	6628716	934001	869	0.093%
fir	DSPstone	37326	12802	691	5.398%
hyper	Trimaran	4746406	485008	738	0.152%
ifthen	Trimaran	9134086	2300001	1506	0.065%
lms	DSPstone	49314	16001	1089	6.806%
matmul	SNU Real-Time	1792224	502501	13661	2.719%
sqrt	SNU Real-Time	1563289	243276	1485	0.610%

that we assume a perfect L1 instruction cache in this work, so that we can focus on analyzing worst-case data cache performance and studying its effect on the WCET. Actually, the stack distance based timing analysis approach can also be applied to the instruction cache, whose addresses are relatively easier to generate. However, the instruction cache timing analysis is out of the scope of this paper.

For this evaluation, we select eight benchmarks from a diverse set of sources, including the SNU real-time benchmark suite [SNU 2009], DSPstone [Zivojnovic et al. 1994] and Trimaran built-in benchmarks [Trimaran 2009]. The benchmarks used in our work are either with the single fixed execution time (e.g. fir, lms, matmul), or with the variable execution time depending on different inputs (e.g. ifthen, hyper, eight,

Table IV. Estimated and Observed Worst-case Data Cache Misses.

Benchmark	Simulated *D* Misses	Estimated *D* Misses	Est./Sim.
dag	1247	1260	1.010
eight	869	879	1.012
fir	691	696	1.007
hyper	738	745	1.009
ifthen	1506	1518	1.008
lms	1089	1092	1.003
matmul	13661	13847	1.014
sqrt	1485	1492	1.005

Table V. Breakdown of Estimated and Observed Data Cache Misses, in Terms of Cold Misses, Conflict Misses and Capacity Misses.

	Simulated Results			Estimated Results			Estimated/Simulated		
Benchmark	Cold	Conflict	Capacity	Cold	Conflict	Capacity	Cold	Conflict	Capacity
dag	879	0	368	885	0	375	1.007	1.000	1.019
eight	629	0	240	637	0	242	1.013	1.000	1.008
fir	402	0	289	404	2	292	1.005	1.000	1.010
hyper	379	0	359	385	0	360	1.016	1.000	1.003
ifthen	1129	0	377	1135	0	383	1.005	1.000	1.016
lms	401	0	688	401	0	691	1.000	1.000	1.004
matmul	940	3021	9700	943	3067	9837	1.003	1.015	1.014
sqrt	505	0	980	511	0	981	1.012	1.000	1.001

dag, sqrt). Since these benchmarks are not too big, we can determine the worst-case causing inputs by analyzing the source codes directly. Also, all the benchmarks are compiled by using the Trimaran compiler. The salient characteristics of the benchmarks are shown in Table III.

4. EXPERIMENTAL RESULTS

4.1 Estimated vs. Simulated Results

Table IV compares the estimated worst-case cache misses and the simulated (i.e., observed) worst-case cache misses for the 8K 2-way set-associative data cache of the baseline processor. We observe that for all the benchmarks, the estimated worst-case number of data cache misses is very close to the simulated worst-case number of data cache misses, indicating that the stack distance based worst-case data cache analysis is safe and highly accurate. Actually, for most benchmarks, the difference between the estimated number and the simulated number of worst-case cache misses is within 1%. On average, the estimated worst-case number of data cache misses is only 0.8% higher than the simulated worst-case number of data cache misses.

An unique advantage of the stack distance based analysis is that it can estimate the data cache misses in terms of cold, conflict and capacity misses. Table V compares the estimated worst-case cold, conflict and capacity misses with the simulated results. The last three columns give the ratios of the estimated cache misses to the simulated cache misses for cold, conflict and capacity misses, which are 100.8%, 100.2% and

Table VI. Estimated and Observed Worst-case Execution Cycles with 10 Cycles Data Cache Miss Penalty.

Benchmarks	Observed WCET	Estimated WCET	Est./Obs.
dag	6431496	6630630	1.031
eight	6628716	6760820	1.020
fir	37326	37398	1.002
hyper	4746406	4945480	1.042
ifthen	9134086	9383210	1.027
lms	49314	49346	1.001
matmul	1792224	1794090	1.001
sqrt	1563289	1658700	1.061

Table VII. Estimated and Observed Worst-case Execution Cycles with 100 Cycles Data Cache Miss Penalty.

Benchmarks	Observed WCET	Estimated WCET	Est./Obs.
dag	6543726	6744030	1.031
eight	6706926	6839930	1.020
fir	99516	100218	1.007
hyper	4812826	5012530	1.041
ifthen	9269626	9519830	1.027
lms	147324	147626	1.002
matmul	3021714	3040320	1.006
sqrt	1696939	1792980	1.057

100.9% respectively on average. These results demonstrate that the proposed approach can safely and tightly estimate all the three different types of data cache misses, which can provide useful insights for cache designers and real-time software developers to enhance the worst-case data caching performance for real-time systems.

4.2 WCET Results

While this paper focuses on worst-case data cache analysis, we have also incorporated the worst-case data cache misses and their latencies into the pipeline timing model based on [Healy et. al 1995] to estimate the WCET for the baseline VLIW processor. Table VI lists the observed worst-case execution cycles (through simulation) and the estimated worst-case execution cycles with 10 cycles data cache miss penalty, and Table VII lists the data with 100 cycles data cache miss penalty. The last column in Table VI and Table VII gives the ratio of the estimated WCET to the observed WCET through simulation. Due to the tight worst-case data cache analysis, which is often one of the most challenging tasks in timing analysis for microprocessors, the estimated WCET is only slightly more than the observed WCET. On average, the estimated WCET is only 2.3% higher than the observed WCET with 10 cycles data cache miss penalty, and 2.4% higher than the observed WCET with 100 cycles data cache miss penalty, which indicates the importance of obtaining precise worst-case data cache performance.

4.3 Sensitivity Results

While our default data cache is 8K and 2-way set-associative, we have also done sensitivity analysis to study the effectiveness of the proposed approach for data caches with different sizes and associativities. Table VIII compares the simulated and estimated cache misses for a 2-way set-associative data cache with its size varying from 4K to 8K and 16K. As we can see, for all these three cache sizes, our approach can compute safe and tight upper bound of the data cache misses. On average, the ratio of the estimated cache misses to the simulated cache misses for a 4K, 8K, or 16K data cache is 100.8%, 100.8%, 101.0% respectively, indicating the effectiveness of the proposed approach for data caches with various sizes.

Table VIII. Estimated and Simulated Worst-case Cache Misses for a 2-way Set-associative Data Cache, with its Size Varying from 4K to 8K and 16K.

	Simulated *D* Misses			Estimated *D* Misses			Estimated/Simulated		
Benchmark	4K	8K	16K	4K	8K	16K	4K	8K	16K
dag	1256	1247	1012	1267	1260	1024	1.009	1.010	1.012
eight	881	869	755	890	879	764	1.010	1.012	1.012
fir	947	691	402	954	696	404	1.007	1.007	1.005
hyper	756	738	379	763	745	387	1.009	1.009	1.021
ifthen	1506	1506	1488	1518	1518	1500	1.008	1.008	1.008
lms	1345	1089	401	1348	1092	401	1.002	1.003	1.000
matmul	22215	13661	2723	22489	13847	2769	1.012	1.014	1.017
sqrt	1508	1485	508	1513	1492	512	1.003	1.005	1.008

Table IX. Estimated and Simulated Worst-case Cache Misses for an 8K Data Cache, with its Set Associativity Varying from 1 Way (i.e., Direct-mapped) to 2 Way, 4 Way and Fully Associative.

	Simulated *D* Misses				Estimated *D* Misses				Estimated/Simulated			
Benchmark	1 way	2 way	4 way	fa	1 way	2 way	4 way	fa	1 way	2 way	4 way	fa
dag	1153	1247	1256	1256	1179	1260	1267	1269	1.023	1.010	1.009	1.010
eight	784	869	869	877	791	879	879	888	1.009	1.012	1.012	1.013
fir	691	691	693	691	710	696	699	699	1.027	1.007	1.009	1.012
hyper	627	738	756	756	655	745	763	765	1.045	1.009	1.009	1.012
ifthen	1537	1506	1506	1506	1557	1518	1518	1521	1.013	1.008	1.008	1.010
lms	977	1089	1089	1089	981	1092	1093	1093	1.004	1.003	1.004	1.004
matmul	47415	13661	15606	16627	47628	13847	15872	16692	1.004	1.014	1.017	1.004
sqrt	1534	1485	1485	1496	2034	1492	1492	1503	1.326	1.005	1.005	1.005

In our next experiment, we fix the data cache size to be 8K, but change its set associativity from 1 way (i.e., direct-mapped) to 2 way, 4 way and fully-associative. Table IX compares the estimated worst-case cache misses with the simulated worst-case cache misses for data caches with different set associativities. As can be seen, generally the estimated worst-case cache misses are close to the simulated worst-case cache misses, especially for set-associative and fully-associative caches. However, for the direct-mapped data cache, we observe that there is a large difference between the estimated results and simulated results, especially for `sqrt`. The reason is that in a direct-mapped cache, the number of conflict misses increases dramatically, and different loop iterations can have quite different numbers of cache misses. Since our approach computes the worst-case data cache misses by adding the largest number of cache misses across all the iterations on different paths, the worst-case path may not be simulated or even be infeasible. Nevertheless, as we can see in Table IX, the average ratio of the estimated worst-case cache misses to the simulated worst-case cache misses of the direct-mapped cache is 105.6%, which is still reasonably accurate as compared to the results reported in state-of-the-art data cache timing analysis research [White et al. 1997; Ramaprasad and Mueller 2005].

Interestingly, we notice that our approach can precisely estimate the worst-case cache misses even for a fully-associative cache. As we can see in the last column of Table IX, the ratio of the estimated cache misses to the simulated cache misses of a fully-associative cache is within 101.3% for all the benchmarks. It is worthy to note that this accurate estimation of the worst-case cache misses for the fully-associative cache does not increase the analysis complexity, since once the worst-case stack distance is calculated, it can be applied to caches with different set associativities to compute or categorize data cache misses without further analysis. This is in contrast to the state-of-the-art data cache timing analysis techniques [White et al. 1997; Li et al. 1996; Ramaprasad and Mueller 2005], whose complexity can aggravate significantly for set-associative caches, especially for fully-associative caches. Therefore, we believe the proposed approach is particularly useful for the data cache design space exploration (i.e., data caches with different sizes and associativities) to enhance the worst-case performance.

5. RELATED WORK

Recently, there have been an increasing number of studies [White et al. 1997; Li et al. 1996; Ferdinand and Wihelm 1998; Ramaprasad and Mueller 2005; Staschulate and Ernst 2006] on statically bounding worst-case data cache performance. Most of these research efforts have focused on examining the worst-case performance for direct-mapped data caches, as the timing analysis of set-associative caches needs to consider both the history of accesses to each cache set and the replacement algorithm, which can dramatically explode the states that need to be maintained and can be prohibitively expensive in terms of the computation time. Although prior work in [White et al. 1997; Li et al. 1996; Ramaprasad and Mueller 2005] has proposed static analysis approaches that can be generally applied to set-associative data caches, all these methods have rather high computational complexity, and no evaluation results are given for set-associative data caches. Therefore, compared with these state-of-the-art techniques, the stack distance based timing analysis approach proposed in this paper can be efficiently applied to both direct-mapped and set-associative caches without increasing the analysis complexity. This can greatly help designers to optimize data cache performance for real-time systems. For example, our approach enables fast cache design space exploration to accurately study the impact of different set associativities on the *worst-case* cache performance, which cannot be efficiently supported by current timing analysis techniques. Furthermore, our approach can classify worst-case data cache misses into cold, conflict and capacity misses without increasing the analysis complexity, which are likely to provide specific and useful insights for improving the worst-case data cache performance.

To enhance the time predictability of data caches, cache locking techniques have been proposed in [Puaut and Decotigny, 2002; Vera et al. 2003]. With cache locking, selected data is loaded into cache and locked in place so that it will not be replaced until it is explicitly unlocked. As a result, the worst-case cache behavior becomes predictable since the cache contents are statically known. Cache locking, however, cannot exploit the dynamic data reuse behavior and thus cannot utilize the cache space efficiently. For example, if a number of cache lines are locked, no other data can use these lines, although they may exhibit certain temporal/spatial locality. This is especially problematic if the data size is much larger than the cache size. Moreover, explicitly locking and unlocking data introduces overheads. The stack distance based data cache timing analysis technique proposed in this paper is orthogonal to the cache locking techniques, and our approach enables the time-predictable data caching for real-time systems without having to employ cache locking.

6. CONCLUSION

This paper proposes a stack distance based approach to computing the worst-case data cache performance. While prior work on stack distance based analysis has focused on studying average-case cache behavior, this paper exploits stack distance to calculate the worst-case data cache behavior by using an efficient path-based analysis technique. An advantage of the stack distance based approach is that it can be applied to both direct-mapped and set-associative caches without increasing the complexity of

analysis. Also, the proposed approach can statically categorize worst-case data cache misses into cold, conflicting and capacity misses, which can provide specific and helpful insights for cache and software designers to optimize the worst-case data cache performance. Our extensive evaluation shows that the proposed stack distance based approach can safely and tightly estimate the worst-case cache performance for a variety of data caches with different sizes and set associativities.

In our future work, we would like to incorporate the infeasible-path-elimination techniques [Healy and Whalley 2002; Chen et al. 2005] to further enhance the accuracy and time efficiency of the stack distance based worse-case data cache analysis. Also, we intend to apply this approach to study the worst-case instruction cache performance, especially for set-associative instruction caches.

ACKNOWLEDGMENT

This work was funded in part by the NSF grants CNS 0720502 and CCF 0914543.

REFERENCES

ARNOLD, R., F. MULLER, D. WHALLEY, AND M. HARMON. Bounding worst-case instruction cache performance. In *Proc. of the Real-Time Systems Symposium*, 1994.

BERG, C., J. ENGBLOM, AND R. WILHELM. 2004. Requirements for an design of a processor with predictable timing. In *Proc. of the Dagstuhl Perspectives Workshop on Design of Systems with Predictable Behavior*.

BEYLS, K. AND E. D'HOLLANDER. 2001. Reuse Distance as a Metric for Cache Behavior. In *Proc. of PDCS'01*, Aug.

CASCAVAL, C. AND D. A. PADUA. 2003. Estimating cache misses and locality using stack distance. *In Proc. of ICS'03*, June.

CHEN, T., T. MITRA, A. ROYCHOUDHURY, AND V. SUHENDRA. 2005. Exploiting branch constraints without exhaustive path enumeration. In *Proc. of the 5th International Workshop on Worst-Case Execution Time Analysis* (WCET) July.

FERDINAND, C. AND R. WILHELM. 1998. On predicting data cache behavior for real-time systems. In *Proc. of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded System.*

HEALY, C. A., D. B. WHALLEY, AND M. G. HARMON. 1995. Integrating the timing analysis of pipelining and instruction caching. In *Proc. of the Real-Time Systems Symposium.*

HEALY, C. AND D. WHALLEY. 2002. Automatic detection and exploitation of branch constraints for timing analysis. *IEEE Transactions on Software Engineering*, 28(8).

http://archi.snu.ac.kr/realtime/benchmark/.

KATHAIL, V., M. SCHLANSKER, AND B. R. RAU. 2000. HPL-PD architecture specification: version 1.1. HPL Technical Report.

KIM, Y. H., M. D. HILLS, AND D. A. WOOD. 1991. Implementing stack simulation for highly-associative memories. In *Computer Sciences Technical Report #997*, Univ. of Wisconsin, February.

LI, Y. S., S. MALIK, AND A. WOLFE. 1996. Cache modeling for real-time software: beyond direct mapped instruction caches. In *Proc. of the IEEE Real-Time Systems Symposium*.

LI, Y. S., S. MALIK, AND A. WOLFE. 1995. Efficient microarchitecture modeling and path analysis for real-time software. In *Proc. of the 16th IEEE Real-Time Systems Symposium*, Dec.

LIM, S., Y. H. BAE, G. T. JANG, B.-D. RHEE, S. R. MIN, C. Y. PARK, AND C. S. KIM. 1994. An accurate worst case timing analysis technique for RISC processors. In *Proc. of the 15th IEEE Real-Time Systems Symposium.*

MUCHNICK, S. S. 1997. Advanced compiler design and implementation. Morgan Kaufmann

Publishers.

PUAUT, I. AND D. DECOTIGNY. 2002. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proc. of 23th Real-Time Systems Symposium* (RTSS'02), Dec.

RAMAPRASAD, H. AND F. MUELLER. 2005. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium*.

ROCHANGE, C. AND P. SAINRAT. 2002. Difficulties in computing the WCET for processors with speculative execution. In *Proc. of WCET*.

STASCHULAT, J. AND R. ERNST. 2006. Worst case timing analysis of input dependent data cache behavior. In *Proc. of the 18th Euromicro Conference on Real-Time Systems* (ECRTS06).

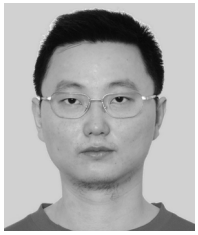
Trimaran homepage, http://www.trimaran.org.

VERA, X., B. LISPER, AND J. XUE. 2003. Data cache locking for higher program predictability. In *Proc. of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*.

WHITE, R., F. MULLER, C. HEALY, D. WHALLEY, AND M. HARMON. 1997. Timing analysis for data caches and set-associative caches. In *Proc. of the IEEE Real-Time Technology and Applications Symposium*, June.

WILHELM, R., J. ENGBLOM, A. ERMEDAHL, N. HOLSTI, S. THESING, D. WHALLEY, G. BERNAT, C. FERDINAND, R. HECKMAN, T. MITRA, F. MUELLER, I. PUAUT, P. PUSCHNER, J. STASCHULAT, AND P. STENSTROM. 2007. The Worst-case execution time problem - overview of methods and survey of tools. In *ACM Transactions on Embedded Computing Systems*, January.

ZIVOJNOVIC, V., J. MARTINEZ, AND C. SCHL. 1994. DSPstone: A DSP-oriented benchmarking methodology. In *Proc. of ICSPAT'94*, Oct..



Yu Liu is currently a PhD student in the Department of Electrical and Computer Engineering of Southern Illinois University Carbondale. He received his B.S and M.S degrees in Sichuan University, China in 2000 and 2003 respectively, and majored in communication engineering. Also, he worked in Motorola as senior software engineer from 2003 through 2007. His research interest includes real-time system, wireless sensor network, and cyber-physical system.

Wei Zhang is an associate professor in Electrical and Computer Engineering at Southern Illinois University Carbondale. He received the B.S. degree in computer science from the Peking University in China in 1997, the M.S from the Institute of Software, Chinese Academy of Sciences in 2000, and the Ph.D. degree in computer science and engineering from the Pennsylvania State University in 2003. His research interests are in embedded and real-time computing systems, computer architecture and compiler. Dr. Zhang has received the 2009 SIUC Excellence through Commitment Outstanding Scholar Award for the College of Engineering, and 2007 IBM Real-time Innovation Award. His research has been supported by NSF, IBM and Altera. He is a senior member of the IEEE. He has served as a member of the technical program committees for several IEEE/ACM conferences and workshops.