Context-Aware Memory Profiling for Speculative Parallelism

Changsu Kim Juhyun Kim Juwon Kang Jae W. Lee† Hanjun Kim
POSTECH Seoul National University†
{kcs9301, memphis, gjw0917, hanjun}@postech.ac.kr jaewlee@snu.ac.kr†

***Abstract*—To expose hidden parallelism from programs with complex dependences, modern compilers employ memory profilers to augment imprecise static analyses. Since dynamic dependence patterns among instructions can vary widely depending on the context, such as function call site stack and loop nest level, *context-aware* memory profiling is of great value for precise memory profiling. However, recording memory dependences with full context information causes huge overheads in terms of CPU cycles and memory space. Existing profilers mitigate this problem by compromising precision, coverage, or both. This paper proposes a new precise Context-Aware Memory Profiling (CAMP) framework that efficiently traces all the memory dependences with full context information. CAMP statically analyzes a context tree of a program that illustrates all the possible dynamic contexts, and simplifies context management during profiling. For 14 programs from SPEC CINT2000 and CINT2006 benchmark suites, CAMP increases speculative parallelism opportunities by 12.6% on average and by up to 63.0% compared to the baseline context-oblivious, loop-aware memory profiler.**

I. Introduction

To aggressively optimize programs, modern compilers [1]–[8] employ memory profilers that trace dynamic memory dependences among instructions. Once the memory profilers identify rarely occurring dependences, the compilers can ignore these dependences and exploit speculative parallelism. For example, even if independence among iterations cannot be proven statically, some compilers optimistically exploit loop-level parallelism when no inter-iteration dependence manifests during profiling [4], [9]–[14]. Therefore, precise, high-coverage profiling techniques are crucial to exploit speculative parallelism most effectively.

Dynamic data dependence patterns can vary widely depending on the program context, such as function call site stack and loop nest level. For example, in nested loops, there may exist an inter-iteration dependence between two instructions in an inner loop, while not in an outer loop. If a memory profiler records dependences without context information, the inter-iteration dependence will be associated with both loops, so neither of them can be parallelized although the outer loop can actually be parallelized. Therefore, context-aware profiling is highly desirable to fully exploit parallelism opportunities.

However, tracing all the memory dependences with their contexts easily become impractical due to its huge overheads in terms of CPU cycles and memory space. Profiling memory dependences greatly increases instruction counts to identify and record dependences between instructions that touch the same memory address. Context awareness exacerbates this problem as dependences between the same pair of instructions are counted separately if their contexts are different. As a result, most of existing memory profilers [15]–[19] trace memory dependences either without contexts or with only partial context information. A few memory profilers log full context information [20], [21], but the profilers compromise its precision by using compacted context information or profiling dependences in a context granularity. Table I summarizes and compares those existing memory profilers.

This paper proposes a new compiler-runtime cooperative Context-Aware Memory Profiling (CAMP) framework that traces memory dependences in a byte level granularity with full context information. The CAMP compiler statically generates a context tree that represents all the possible dynamic contexts, encodes every context in a single context ID and its static offset, and provides the CAMP runtime with the offset as a hint. The CAMP runtime restores the dynamic context ID with one arithmetic operation between the current context ID and the given static offset, and records memory access history with the context ID. This context encoding simplifies the data structure and algorithm of CAMP, and minimizes its profiling time and memory overheads.

This work implements CAMP on top of the LLVM compiler framework [22]. For 14 programs from SPEC CINT2000 and CINT2006 benchmark suites, CAMP exposes average 12.6% and up to 63.0% more speculative parallelism opportunities than a context-oblivious, loop-aware memory profiler (LAMP). Here, CAMP finds that 87.6% of memory dependences in the LAMP results are false positive, with only 47.2% and 28.0% of additional profiling time and memory usage.

In summary, the primary contributions of this paper are:

- A compiler-runtime cooperative context-aware memory profiling system with full contexts
- A static context tree that represents all the possible dynamic contexts such as call site stack and loop nest
- An in-depth evaluation of CAMP using 14 benchmarks from SPEC CINT2000 and CINT2006 benchmark suites

II. Motivation

Memory profiling results about dynamic dependences enable modern compilers [1]–[8] to support aggressive optimization that cannot be achieved by static analyses only. For example, automatic speculative parallelizing compilers [4], [9]–[14], [23] collect dynamic dependences in loops, and speculatively parallelize the loops ignoring rarely occurring dependences that static analysis cannot remove. Moreover, memory dependence profiling helps parallelizing compilers produce robust codes by augmenting fragile static analyses [23].

TABLE I
COMPARISON OF MEMORY PROFILING SYSTEMS

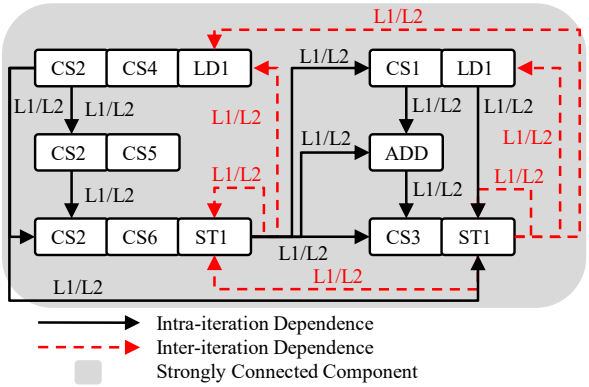
System	Context-Awareness		Full Coverage of Dependences	Whole Program Coverage	Profiling Granularity
	Loop-Awareness	Call Site-Awareness			
H. Yu et al. [19]	✓	×	✓	×	Variable
A. Ketterlin et al. [16]	✓	×	✓	✓	Variable
R. Vanka et al. [18]	✓	×	×	✓	Byte
M. Kim et al. [17]	✓	×	✓	✓	Byte
T. Chen et al. [20]	✓	Compacted Call Path	×	✓	Byte
Y. Sato et al. [21]	✓	✓	×	✓	Context
CAMP [This paper]	✓	✓	✓	✓	Byte

```
1  int getValue(Node n) {
2    return n.value;                  // LD1
3  }
4
5  void setValue(Node n, int v) {
6    if(isValid(v))                    // CS7
7      n.value = v;                   // ST1
8  }
9
10 int work(Node n) {
11   int v1 = getValue(n);            // CS4
12   int v2 = update(v1);             // CS5
13   setValue(n, v2);                 // CS6
14   return v2;
15 }
16
17 void main() {
18   for(int t = 0; t < T; t++) {     // L1
19     for(int i = 0; i < N; i++) {   // L2
20       int s = getValue(sum[t]);    // CS1
21       int v = work(nodes[i]);      // CS2
22       s = s + v;                   // ADD
23       setValue(sum[t], s);         // CS3
24     }
25   }
26 }
```

Fig. 1. Example program

```
LD1 -> ST1 @L1(Intra-Iteration)
ST1 -> LD1 @L1(Inter-Iteration)
ST1 -> LD1 @L1(Intra-Iteration)
ST1 -> ST1 @L1(Inter-Iteration)
ST1 -> ST1 @L1(Intra-Iteration)
LD1 -> ST1 @L2(Intra-Iteration)
ST1 -> LD1 @L2(Inter-Iteration)
ST1 -> ST1 @L2(Inter-Iteration)
```

(a) Loop-aware (context-oblivious) profiling results



(b) Speculative PDG with loop-aware profiling results

Fig. 2. A loop-aware profiling result and a simplified speculative program dependence graph of the example program in Figure 1.

The dynamic dependences manifest depending on their contexts such as call site stacks and loop nest levels. Figure 1 shows a simple example program that iterates two arrays in a nested loop. `LD1` in `getValue` and `ST1` in `setValue` load and store `values` of the two linked lists such as `sum` and `nodes`. Since `sum[t]` is loop invariant for loop `L2`, `LD1` of `CS1` (`LD1@CS1`) and `ST1` of `CS3` (`ST1@CS3`) repeatedly access the same `value` in `L2`, yielding an inter-iteration dependence from `ST1@CS3` to `LD1@CS1`. However, since `nodes[i]` is changing for `L2`, `LD1@CS4@CS2`s and `ST1@CS6@CS2`s of different iterations do not access the same `value` nor generate any inter-iteration dependence from `ST1@CS6@CS2` to `LD1@CS4@CS2` for `L2`.

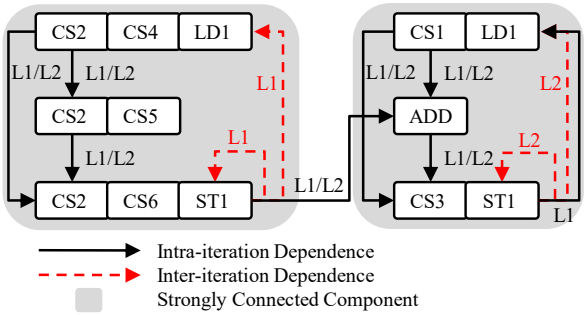
Since most existing memory profilers [15]–[19] are aware of only loop nest levels, the loop-aware memory profilers can limit the applicability of speculative parallelization. Since the loop-aware memory profilers are oblivious of call site stacks, their profiling results do not include any call site information like Figure 2(a). Therefore, with the loop-aware profiling results, a compiler cannot differentiate instructions from different call sites such as `LD1@CS1` and `LD1@CS4@CS2`, and generates a less precise speculative program dependence graph (PDG) like Figure 2(b) that includes many false positive memory dependences like an inter-iteration dependence from `ST1@CS3` to `LD1@CS4@CS2`. Since the less precise speculative PDG forms only one strongly connected component, the compiler cannot parallelize the nested loops in the example.

To more aggressively optimize the program, the compiler needs profiling information with full contexts such as not only loop nest levels but also call site stacks. As Figure 3(a) shows, a context-aware memory profiler records memory dependences with their contexts, and allows a compiler to generate different dependences for the same instructions with different contexts. As a result, the compiler generates a precise context-aware speculative PDG with two strongly connected components like Figure 3(b), and parallelizes the nested loops like Figure 4.

The increased parallelism opportunities from the context-aware profiling lead to the additional performance speedups

```
LD1@CS1      -> ST1@CS3      @L1(Intra-Iteration)
LD1@CS4@CS2  -> ST1@CS6@CS2  @L1(Intra-Iteration)
ST1@CS6@CS2  -> LD1@CS4@CS2  @L1(Inter-Iteration)
ST1@CS6@CS2  -> ST1@CS6@CS2  @L1(Inter-Iteration)
ST1@CS3      -> LD1@CS1      @L1(Intra-Iteration)
ST1@CS3      -> ST1@CS3      @L1(Intra-Iteration)
LD1@CS1      -> ST1@CS3      @L2(Intra-Iteration)
LD1@CS4@CS2  -> ST1@CS6@CS2  @L2(Intra-Iteration)
ST1@CS3      -> LD1@CS1      @L2(Inter-Iteration)
ST1@CS3      -> ST1@CS3      @L2(Inter-Iteration)
```

(a) Context-aware profiling results



(b) Speculative PDG with context-aware profiling results

Fig. 3. A context-aware profiling result and a simplified speculative program dependence graphs of the example program in Figure 1.

```
1  void main() {
2    for(int t = 0; t < T; t++) {              // L1
3      parallel_for(int i = 0; i < N; i++) {   // L2
4        int v[t][i] = work(nodes[i]);         // CS2
5      }
6    }
7
8    parallel_for(int t = 0; t < T; t++) {     // L1'
9      for(int i = 0; i < N; i++) {            // L2'
10       int s = getValue(sum[t]);             // CS1
11       s = s + v[t][i];                      // ADD
12       setValue(sum[t], s);                  // CS3
13     }
14   }
15 }
```

Fig. 4. Parallelized program for `L1` and `L2` in Figure 1

of speculative parallel program. Figure 5 shows the performance impact of the context-aware profiling for speculatively parallelized programs in SPEC CINT2000 and CINT2006 benchmark suites [24]. The programs are manually parallelized with a distributed multi-threaded transactional memory [25] on a 128-core cluster according to Spec-DSWP parallelization algorithm [26]. While speculative parallelization with loop-aware profiling results achieves a geomean speedup of 8.1×, speculative parallelization with context-aware profiling results increases parallelism opportunities for `164.gzip`, `197.parser`, and `256.bzip2`, and achieves a geomean speedup of 26.8×.

Although context-aware memory profiling enables more aggressive optimization and additional performance speedups, it severely increases profiling time and memory usage. Whenever a program executes a load or store instruction on a memory location, a memory profiler generates dependences between the

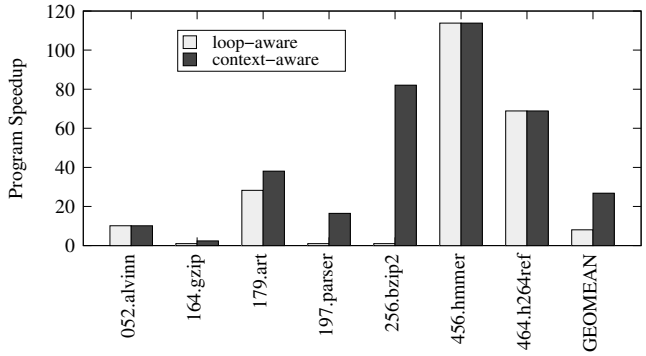


Fig. 5. Program speedups from speculative parallelization with loop-aware and context-aware profiling results

instruction and all the previous instructions that accessed the same memory location. While loop-aware memory profilers record only nested loops and their iteration numbers for each memory instruction, the context-aware memory profiler should additionally record call site stacks. Moreover, the context-aware memory profiler should handle the same instruction with different call site stacks as different instruction instances, requiring additional profiling operations and memory spaces.

Due to the high profiling overhead, most of the existing memory profilers [15]–[19] do not collect full contexts of memory dependences. Although T. Chen et al. [20] and Y. Sato et al. [21] propose context-aware memory profilers that collect call site stacks and loop nests, there are false positive dependences in their results because of coarse granularity [21] and abstracted transitive dependence recording [20]. Unlike the existing memory profilers, the CAMP framework reduces the profiling overhead with the compiler-assisted context management, and supports byte-level context-aware memory profiling without significant performance and memory overheads. Table I summarizes and compares the existing memory profilers.

III. Compiler-assisted Context Management

While Section II points out the necessity of context-aware memory profiling, managing contexts of a program requires huge profiling time and memory overheads. To simplify context management of CAMP , the CAMP compiler statically analyzes all the possible contexts of a program with a context tree (Section III-A), encodes the contexts into a single integer (Section III-A), and automatically inserts context management instructions (Section III-B).

A. Static Context Tree and Encoding

To efficiently manage contexts and their changes, the CAMP compiler statically analyzes all the possible contexts and encodes each context into a single integer context ID and its offset.

First, the compiler generates a static context tree that represents all the possible contexts of a program. The compiler creates the static context tree by recursively inserting a child node for every loop nest and function call site. Figure 6 shows

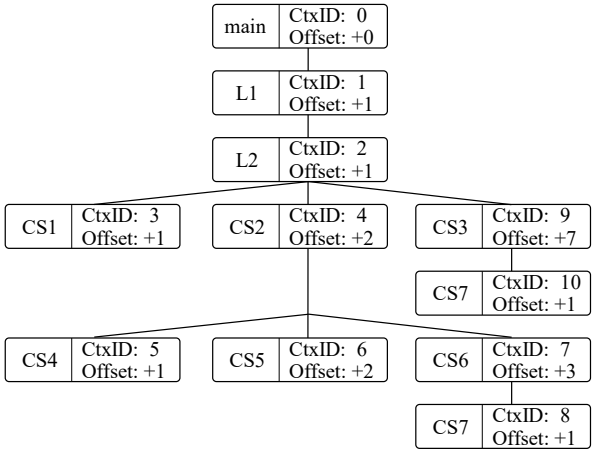
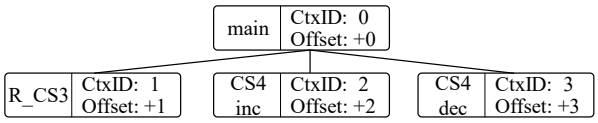


Fig. 6. Context tree for the example code in Figure 1

```
1  bool is_even(unsigned int n) {
2    if (n==0) return true;
3    else return is_odd(n-1);  // CS1
4  }
5
6  bool is_odd(unsigned int n) {
7    if (n==0) return false;
8    else return is_even(n-1); // CS2
9  }
10
11 int inc(int n) { return n+1; }
12
13 int dec(int n) { return n-1; }
14
15 void main() {
16   int (*fPtr)(int);
17   if(is_even(n))              // CS3
18     fPtr = &inc;
19   else
20     fPtr = &dec;
21   n = fPtr(n);                // CS4
22 }
```

(a) Code example with recursive and indirect function calls



(b) Context tree

Fig. 7. Context tree for recursive and indirect function call

a context tree for the example code in Figure 1. The compiler recursively inserts `L1` and `L2` for each loop of the `main` node. Then, the compiler inserts three different children such as `CS1`, `CS2` and `CS3` for each call site of the `L2` node, and recursively inserts three children such as `CS4`, `CS5` and `CS6` for `CS2`. Here, though call sites `CS1` and `CS4` call the same function `getValue`, the compiler inserts different context nodes for them. As a result, the context tree can differentiate the same memory instructions across all the different dynamic contexts like `LD1` of `CS1` and `LD1` of `CS4`. Here, unlike the existing context trees such as Loop Call Context Tree (LCCT) [21], [27] and Call Context Tree (CCT) [28]–[30] that profilers dynamically generate at profiling time, the CAMP compiler statically generates the context tree to alleviate context management runtime overheads of the memory profiler.

At the context encoding step, the CAMP compiler encodes each context node in the context tree into a single context ID. Since a loop invocation and a call site instruction can be multiple context nodes in the context tree with different context IDs, the compiler needs to assign different context IDs for the same loop and call site. Addressing the problem, the compiler encodes the contexts in a form of a unique path sum where each loop invocation and call site have the same static offset, and assigns the context IDs and their static offsets with pre-order tree traversal. Figure 6 shows context IDs and their static offsets for the context tree. Though `isValid` is invoked in multiple contexts with different IDs such as `8` and `10`, its static offset from its parent contexts is one value, `+1`. The CAMP runtime dynamically calculates the context IDs by adding the static offset to the current context ID.

While the CAMP compiler creates a context tree for most cases, there are two special cases that require special manipulation; recursive function call and indirect function call. To prevent from generating a context tree infinitely, the CAMP compiler marks recursive functions before generating the context tree, and inserts only the first recursive function call site as a leaf *loop* context node. Here, the compiler considers the recursive function call site node as a loop node, so the CAMP profiler can find its recursion depth by counting iteration numbers. For indirect function calls, the compiler analyzes all the possible targets among referenced functions and inserts the targets as children nodes. Figure 7(a) shows an example code with recursive function calls and indirect function calls. Figure 7(b) illustrates that the compiler adds call site `CS3` as a leaf node due to recursive function calls between `is_even` and `is_odd`, and inserts all the possible indirect call candidates such as `inc` and `dec` for call site `CS4`.

B. Context Management Code Generation

During the program execution, contexts such as call site stacks and loop nests are continuously changing. To reduce context management overheads, the CAMP compiler statically finds the context changing points such as entries and exits of the functions, loop invocations and loop iterations, and inserts instructions to notify the CAMP runtime of the context changes. Followings are the context changing notifiers, and Figure 8 shows how the CAMP compiler inserts the notifiers to the example in Figure 1.

- **change_context(offset)** notifies the change of a call site stack to the CAMP profiler with `offset`. The profiler calculates the new context ID by adding the offset to the current context ID.
- **begin/end_loop(offset)** notifies the begin and the end of a loop to the profiler with `offset`. If a loop begins, the profiler pushes an iteration counter to the iteration counter stack that has iteration counts of loop nests. If a loop

```
1  int getValue(Node *n) {
2    profiling_load(&(n->value), LD1);
3    return n->value;                  // LD1
4  }
5
6  void setValue(Node *n, int v) {
7    change_context(+1);
8    bool t = isValid(v);              // CS7
9    change_context(-1);
10   if(t) {
11     profiling_store(&(n->value), ST1);
12     n->value = v;                   // ST1
13   }
14 }
15
16 int work(Node *n) {
17   change_context(+1);
18   int v1 = getValue(n);             // CS4
19   change_context(-1);
20   change_context(+2);
21   int v2 = update(v1);              // CS5
22   change_context(-2);
23   change_context(+3);
24   setValue(n, v2);                  // CS6
25   change_context(-3);
26   return v2;
27 }
28
29 void main() {
30   begin_loop(+1);
31   for(int t = 0; t < T; t++) {      // L1
32     begin_loop(+1);
33     for(int i = 0; i < N; i++) {    // L2
34       change_context(+1);
35       int s = getValue(sum[t]);     // CS1
36       change_context(-1);
37       change_context(+2);
38       int v = work(nodes[i]);       // CS2
39       change_context(-2);
40       s = s + v;                    // ADD
41       change_context(+7);
42       setValue(sum[t], s);          // CS3
43       change_context(-7);
44       next_iteration();
45     }
46     end_loop(-1);
47     next_iteration();
48   }
49   end_loop(-1);
50 }
```

Fig. 8. Transformed program by the CAMP compiler for the program in Figure 1. Bold lines are added by the CAMP compiler.

context ends, the profiler pops the iteration counter from the iteration counter stack.

- **next_iteration()** notifies the iteration change to the CAMP profiler. The profiler increases the iteration counter at the top. Since only the iteration counter at top of the stack (i.e., the inner most loop) can iterate, no argument is necessary.
- **profiling_load/store(address, instruction ID)** notifies the memory load and store to the CAMP profiler. The profiler updates memory access history and generates dependences for the memory address and the instruction.

With the context changing notifiers, the CAMP runtime reflects context changes and updates context IDs during the program execution. Since the most recently called function returns first, and the most recently entered loop (inner-most loop) exits first, programs change their contexts following the LIFO rule. As a result CAMP manages the dynamic context ID by adding or subtracting the context offset into and from the current context ID according to the context changes. For example, when a program enters (exits) a function, the context manager adds (subtract) the context offset into (from) the current context ID. To manage dependences in the iterated contexts like nested loops, CAMP has the iteration counter stack that keeps how many times each loop nest iterates. When a program enters (exits) a loop nest, the context manager pushes (pops) an iteration counter into (from) the current iteration counter stack. For every loop iteration, CAMP also changes the iteration information in the iteration counter.

IV. Context-Aware Memory Profiling Runtime

Figure 9 illustrates the overall structure of the CAMP runtime. The CAMP runtime mainly consists of the three components: current context, dependence table and history table. This section describes the overall algorithm of context-aware memory profiling and each component of the runtime in details.

A. Algorithm of Context-Aware Memory Profiling

A dynamic instruction instance has its context that represents function call site stacks and iteration information of nested loops when the instruction is executed. When CAMP generates a dependence, CAMP needs to keep the instruction context into a dependence context that represents the context where the dependence is valid. For example, an inter-iteration RAW dependence from `ST1@CS3(Ctx9)` to `LD1@CS1(Ctx3)` in Figure 3(b) is valid at `Loop L2`, but is not valid at `Loop L1`. On the other hand, an inter-iteration RAW dependence from `ST1@CS6@CS2(Ctx7)` to `LD1@CS4@CS2(Ctx5)` is valid at `Loop L1`, but is not valid at `Loop L2`. Therefore, when adding a dependence, CAMP should record its valid contexts such as `Ctx3` and `Loop L2`.

Algorithm 1 describes how the CAMP runtime generates dependences with valid contexts. First, the CAMP runtime updates the dependence table (Lines 1-15). When an instruction accesses a memory location, the runtime receives the memory address (`addr`) and instruction ID (`dstID`), and has the current context ID (`dstCtx`) and iteration counts of nested loops (`dstIterStack`). The runtime searches previous memory instructions (`srcID`, `srcCtx` and `srcIterStack`) that access the same memory address from its history table (Line 1). Then, the runtime inspects the existence of the same dependence in the dependence table to avoid creating redundant dependences (Line 2). If there exists the same dependence with the same context in the dependence table, the runtime updates the existing dependence. If there is no same dependence, the runtime newly generates a dependence with the instruction IDs and their context IDs. The CAMP runtime calculates iterative relation of the dependence by comparing each iteration count in the source and destination iteration stacks (Lines

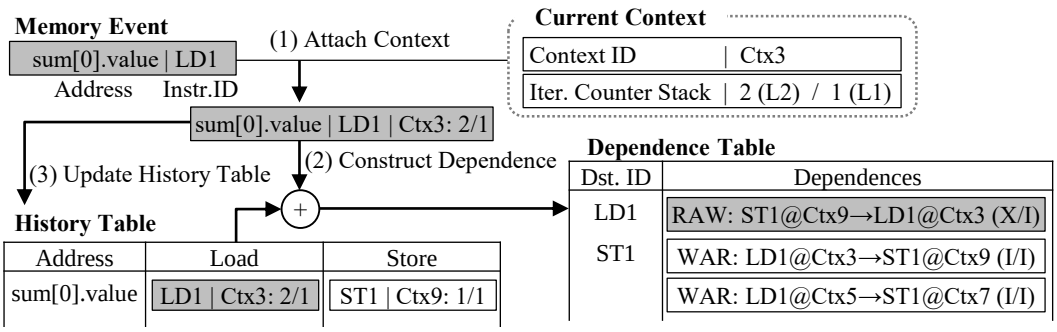


Fig. 9. Structure of the CAMP runtime and its operation example on the program in Figure 1. In the memory event with the context, the numbers after the context ID (Ctx3) are iteration counts of nested loops. In the dependence table element, the right-most values indicate loop iteration relation (I and X mean 'INTRA' and 'INTER', respectively). Updated elements by the operations in the figure are shaded in grey.

Algorithm 1: Context-aware Dependence Generation

```
   Data: addr: accessed address
   Data: dstID: accessed instruction ID
   Data: dstCtx: current context ID
   Data: dstIterStack: current iteration stack
   /* Update Dependence Table                                   */
1  foreach (srcID, srcCtx, srcIterStack) ∈ getHistory(addr) do
2      let dep = getDependence(srcID, srcCtx, dstID, dstCtx);
3      if dep == NULL then
4          dep = createDep(srcID, srcCtx, dstID, dstCtx);
5      end
6      let depIter = getDependenceIter(dep);
7      foreach level = 0 to
       minStackLevel(srcIterStack,dstIterStack) do
8          if srcIterStack[level] == dstIterStack[level] then
9              depIter[level] |= INTRA;
10         else
11             depIter[level] |= INTER;
12             break;
13         end
14     end
15 end
   /* Update History Table                                      */
16 if dstID == STORE then
17     replaceElement(addr, dstID, dstCtx, dstIterStack);
18     clearLoadHistory(addr);
19 else
20     addElement(addr, dstID, dstCtx, dstIterStack);
21 end
```

6-14). If the iteration counts are the same, the dependence is an intra-iteration dependence. If the counts are different, the dependence is an inter-iteration dependence. Since the iterative relation is valid only in the same loop invocation, the runtime stops the comparison if the iteration counts of the two instructions are different. Here, the CAMP runtime keeps all the previous iteration relations, so one dependence can have both inter- and intra-iteration relations.

After updating the dependence table, the CAMP runtime updates the history table with the new instruction. (Lines 16-21). If the current instruction is a load, the runtime simply adds the current instruction and its context in the load history table. However, if the current instruction is a store, the runtime does not only replace the element in the store history table with the current instruction and its context, but also clears elements in the load history table because WAR dependence is the relation between the current store and all the previous loads after the last store instruction.

B. Memory Event with Context

In addition to context changing notifiers, the CAMP compiler finds all the memory related instructions such as loads, stores, memory allocation, memory deallocation and memory sets, and inserts the instructions to notify the CAMP runtime of execution of the memory related instructions. To efficiently manage the dependence table, the compiler statically and sequentially assigns numbers to all the load and store instructions. Since the compiler knows the total number of load and store instructions, the runtime can allocate an array for the dependence table and use the ID as an index.

Figure 9(1) shows how the CAMP runtime creates a memory event context from the memory event and the current context for the example code in Figure 8 and the context tree in Figure 6. When `Instruction LD1` in `getValue` called by `CS1` accesses `value` when `t` is `0` and `i` is `1`, `profiling_load(&(n->value), LD1);` notifies the memory event with a memory instruction (`LD1`) and its memory address (`sum[0].value`). The runtime merges the instruction with context ID (`Ctx3`) and iteration counters (`2/1`), and generates a memory event context as `sum[0].value|LD1|Ctx3:2/1`. The generated memory event context will be used in the history table and dependence generation.

C. Dependence Table

While executing programs, the CAMP runtime directly generates RAW, WAR and WAW dependences and records the dependences in the dependence table. Since the CAMP compiler lets the CAMP runtime know the total number of load and store instructions, the runtime allocates the dependence table as an array indexed by destination ID. Since different dependences can share the same destination instruction, multiple dependences can be stored in each element in the dependence table, so the runtime uses linked lists for each destination.

TABLE II
BENCHMARK DETAILS. K AND M IN THE NUMBERS OF DYNAMIC INSTANCES MEAN THOUSANDS AND MILLIONS

Benchmark	# of Static Instances					# of Dynamic Instances				# of P'll Loops	
	Functions	Loops	Call Sites	Loads	Stores	Calls	Loop Invo.	Loads	Stores	LAMP	CAMP
052.alvinn	9	39	2	261	128	139K	278K	1338K	6715K	15	15
164.gzip	70	200	462	1191	1134	83M	35M	2368M	522M	24	29
175.vpr	155	482	2299	4250	1336	113M	50M	1616M	573M	221	226
177.mesa	1019	1340	4827	16594	11744	3913M	8M	4188M	3172M	41	41
179.art	26	132	274	674	282	14M	30M	578M	314M	55	58
197.parser	323	883	1066	3786	1375	193M	104M	1388M	400M	78	102
256.bzip2	74	253	239	1215	864	54M	36M	5709M	1157M	29	34
300.twolf	190	1082	2294	10585	3773	12M	20M	407M	125M	343	374
401.bzip2	69	301	487	2514	1662	41M	101M	1116M	267M	54	88
433.milc	235	329	2680	3498	1064	216M	31M	6717M	2025M	64	76
456.hmmer	467	1124	5168	9739	4594	47M	47M	3392M	1853M	175	177
462.libquantum	95	119	568	646	345	182M	77M	5366M	2089M	26	26
464.h264ref	948	2608	3521	44386	14342	12M	18M	913M	148M	172	173
470.lbm	19	44	87	253	104	3M	53K	52M	30M	41	44

Figure 9(2) shows how the CAMP runtime generates a RAW dependence and stores the dependence in the dependence table from the example code in Figure 8. Given the memory event context (`sum[0].value|LD1|Ctx3:2/1`), the runtime looks up the history table for the same address, and finds a store context (`ST1|Ctx9:1/1`). With the two memory event contexts, the runtime generates a context-aware dependence according to Algorithm 1. Since the iteration counts are different at `L2`, the profiler marks an inter-iteration dependence (marked as `X`) for `L2`. Since the iteration counts are the same at `L1`, the profiler marks an intra-iteration dependence (marked as `I`) for `L1`.

D. History Table

The CAMP runtime has load and store history tables that keep previously accessed load and store instructions for each memory location. Whenever a memory instruction accesses a memory location, the runtime looks up the access history from the history tables, generates dependences between the current instruction and all the previous instructions in the history tables, and updates the history tables with the current instruction. Since the CAMP runtime executes these operations for every memory accesses, designing efficient history tables and management algorithm is crucial for profiling performance. To efficiently manage the history tables, the runtime allocates each entry of history table at *shadow memory* which memory address is a result of a few bit operations on the accessed memory address. For example, if the memory address of `sum[0].value` is `0x00ACCE55`, the runtime allocates its corresponding history element at `0x8ACCE550`, so two bit operations such as one OR and one SHIFT are enough for the runtime to access the history element.

Figure 9(3) shows how the CAMP runtime updates the history table on the memory event. After updating the dependence table, the runtime updates the history table with the new memory event. If the current memory event is a memory load like `LD1|Ctx3:2/1`, the runtime simply adds the instruction context in the load history table. Thus, there can exist more than one load instruction context for the same memory address in the load history table. However, if the current memory event is a memory store, the runtime replaces the element in the store history table to the instruction context, so there exists at most one instruction context for each memory address in the store history table. Moreover, a store memory event clears elements in the load history table. This clearance allows the runtime not to generate false positive WAR dependences between the current store instruction and a load instruction before the previous store instruction.

V. EVALUATION

We implemented the CAMP framework on top of the LLVM compiler infrastructure [22]. The framework is evaluated with 14 general-purposed programs in the SPEC CINT2000 and CINT2006 benchmark suites [24]. All the evaluations were done natively on an Intel® Core™ i7-4770 machine that has 4 cores running at 3.40GHz and 16 GB of RAM. The programs were compiled with the -O3 optimization flag.

Table II lists the evaluated programs along with information such as brief description and statistics on static and dynamic profiled contexts and memory instructions. Details about each program can be found in [24]. The numbers of loops and call sites in the programs vary from 41 (`052.alvinn`) to 6,292 (`456.hmmer`), and the numbers of dynamic memory instructions also vary from 8.0 millions (`052.alvinn`) to 8.7 billions (`433.milc`).

A. Impact on Speculative Parallelism

To evaluate how effective CAMP is to increase parallelism opportunities, this work compares the numbers of speculatively parallelizable loops by CAMP and a context-oblivious, loop-aware memory profiler (LAMP). Table II shows the numbers of speculatively parallelizable loops by CAMP and LAMP, and Figure 10 illustrates their increase ratios. Compared to

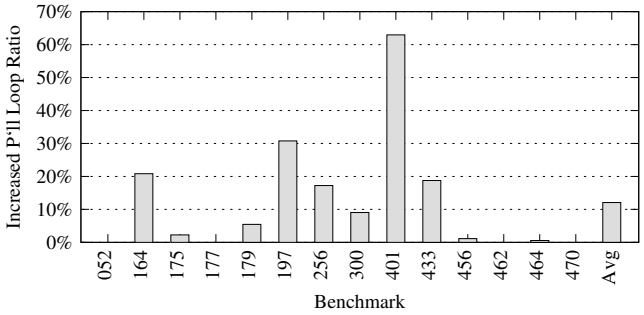


Fig. 10. Increased ratio of parallelizable loops compared to context-oblivious, loop-aware memory profiling results

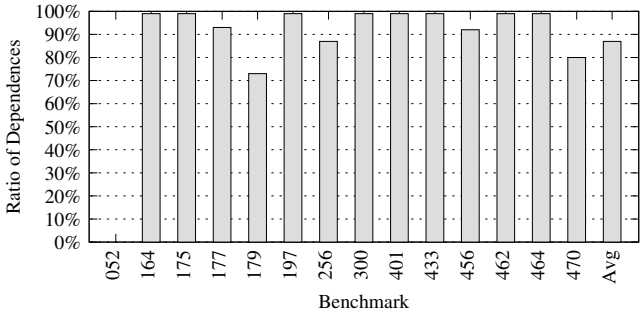
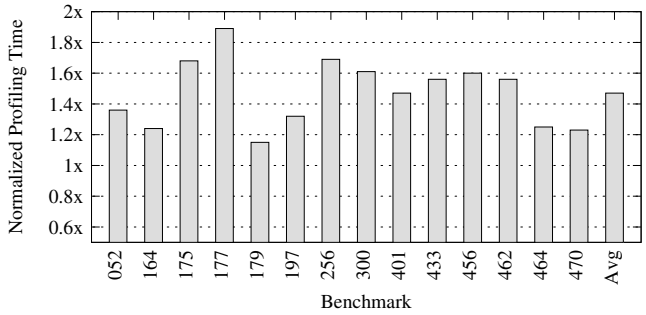
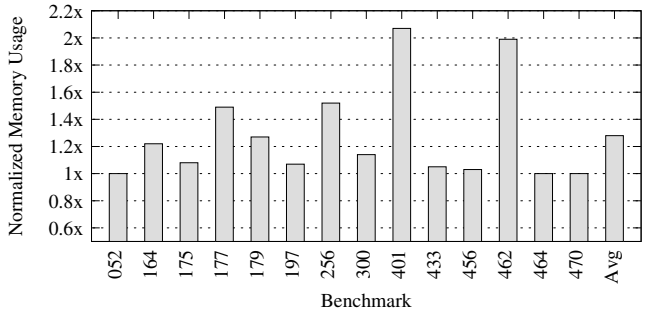


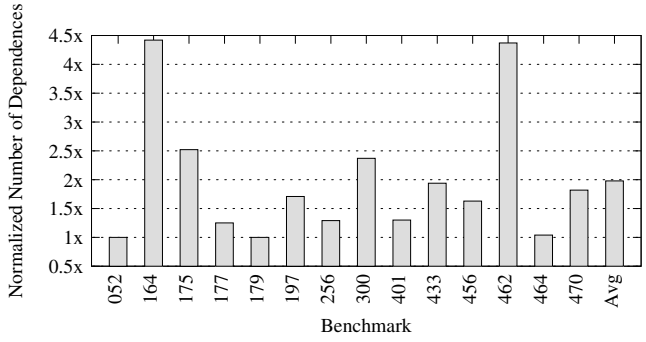
Fig. 11. Ratio of false positive dependences that CAMP finds from context-oblivious, loop-aware memory profiling results



(a) Profiling time



(b) Memory usage



(c) Number of profiled dependences

Fig. 12. Profiling time, memory overheads and numbers of profiled dependences normalized to a context-oblivious, loop-aware memory profiler.

LAMP, CAMP increases the number of speculatively parallelizable loops by 12.6% on average and by up to 63.0% (`401.bzip2`). CAMP cannot increase parallelism opportunities for `052.alvinn`, `177.mesa` and `462.libquantum` because the programs have regular memory access patterns for LAMP enough to find parallelizable loops.

Modern compilers [1]–[8] parallelizes a program relying on program dependence graphs (PDGs), and CAMP allows the compilers to distinguish memory accesses at different call sites and loop nests to generate precise speculative PDGs. To deeply analyze how CAMP increases the speculative parallelism opportunities, this work evaluates ratios of false positive dependences that CAMP finds from context-oblivious, loop-aware memory profiling results. Here, the false positive dependences are the dependences that do not manifest but a loop-aware memory profiler marks as manifest.

Figure 11 shows that CAMP finds that 87.6% of loop-aware memory profiling results for 14 programs are false positive. Since the programs manipulate memory values through getter and setter functions across all the program points, the compilers conservatively insert dependences for all the combinations between getters and setters, increasing false positive dependences. Unlike the context-oblivious, loop-aware memory profilers, since CAMP can collect memory access history with call site contexts, the compilers can find getter and setter functions with consecutive memory accesses on the same memory space, so reduce the number of combinations, yielding precise speculative PDGs. For example, only for the variable `outcnt` in `164.gzip`, the context-aware memory profiler finds 1,344,238 false positive dependences while there exist 499 real memory dependences. Considering that imprecise PDGs limit optimization opportunities of modern compilers, the context-awareness allows the compilers to generate precise speculative PDGs and increase optimization performance.

Though CAMP cannot distinguish call sites and loop nests in the recursive calls, CAMP creates effective context trees and generates precise profiling results for programs with recursive function calls such as `177.mesa`, `300.twolf`, `456.hmmer` and `462.libquantum`. For example, CAMP still increases parallelism opportunities by 9.0% for `300.twolf` marking 99.9% of dependences in the LAMP results as false positive.

B. Time and Memory Overheads of CAMP

Figure 12 shows normalized profiling time, memory overheads and numbers of profiled dependences of CAMP. Bases are those of the context-oblivious, loop-aware profiling (LAMP). Compared to LAMP, CAMP increases the profiling time and memory usage by 47.2% and 28.0% respectively. Most of the increased overheads come from generating additional dependences between the same instructions with different contexts that LAMP cannot distinguish, while context management overheads are negligible. Here, CAMP efficiently manages memory access history using the shadow memory-based history tables with two bit operations, so CAMP minimizes the increased profiling time and memory overheads less than 89.2% while CAMP generates 1.98× more dependences than LAMP.

CAMP generates a large number of additional dependences for `164.gzip` and `462.libquantum` because the programs frequently use wrapper functions such as buffer utility functions and `quantum_` library functions in different locations. Since the buffer utility functions and `quantum_` library functions update global variables, and the programs update global variables through the functions, the same memory access instruction can have a large number of different contexts, increasing the number of context-aware dependences. Moreover, the graphs in Figure 12 show that there is almost no correlation between the profiling overheads and the additional dependence generations, proving CAMP efficiently adds additional dependence to the dependence table.

VI. Related Work

Context-Aware Memory Profilers: Like CAMP, context-aware memory profilers [20], [21] generate memory dependences with their contexts such as function call stacks and loop nests. However, none of them fully generates the memory dependences between all the instructions in a program.

T. Chen et al. [20] made full-transitive data dependence profiler using a unified load/store history table. Since the history table only records the most recent memory instruction, the profiler only generate dependences between the current memory instruction and the most recent memory instruction on the same memory, and the compiler reconstructs full memory dependences from the profiling results with transitive relationship. However, since an instruction can touch multiple memory addresses, the reconstruction can generate false positive results. Moreover, while the profiler uses the expensive hash function to access the elements in the table. CAMP accesses the elements in the history tables in a few bitwise operations that require much less performance overhead.

Y. Sato et al. [21] generated dependences between code regions such as loops and functions instead of instructions. Since there is no information about dependences between instructions in the profiling results, the profiling results limit speculative compiler optimization. Moreover, the profiler only generate RAW dependences, so reordering instructions without renaming is limited.

Loop-Aware Memory Profilers: Loop-aware memory profilers [15]–[19] trace memory dependences only with loop contexts. Although the profilers find inter-iteration and intra iteration dependences like CAMP, they cannot distinguish dependences from different function call stacks.

J. R. Larus [15] proposed automatic parallelization system using a loop-aware memory profiler. The system checks inter-iteration dependences but does not check intra-iteration dependences. Due to its inefficient memory access history management, the profiler suffers from severe memory overhead and time overhead.

M. Kim et al. [17] proposed SD3 profiler that is a parallel memory profiler. SD3 reduces profiling time overhead with the parallel profiling, and also reduces memory usage overhead with data compression using frequent loop-stride characteristic of computational program. Since each memory dependence generation in CAMP is independent each other if they access different memory address, CAMP also can be parallelized like SD3, and additional profiling overhead reduction can be achieved.

H. Yu et al. [19] proposed an object-based dependence profiler. The profiler attaches tags to variables that have access history on the variables. This work profiles a target loop instead of the whole program, so users should execute the profiler multiple times to optimize multiple regions in a program.

R. Vanka et al. [18] proposed a set-based dependence profiler using software signatures. The profiler statically finds relevant dependences that are required for optimization, and profiles the instructions. Although the profiler has low time overhead, the profiling results can be incorrect because the profiler profiles only pre-selected instruction sets.

A. Ketterlin et al. [16] optimized profiling overhead using two main techniques: coalescing consecutive accesses and parameterizing loop nests. The profiler treats consecutive data structures like arrays as a single entity. In other words, the profiler supports variable profiling granularity for consecutive data structure. Parameterizing loop nests reduces profiling overheads exploiting static control loops where all the memory accesses are determined only by parameters of the loops.

Context Management in Profilers: Context management of CAMP is highly inspired by previous context-aware performance profilers [28]–[30] and calling context encoding [31], [32]. G. Ammons et al. [28] first introduced a call tree in which each node reflects a call site. Adaptive calling context tree profilers [29], [30] support sampling-based calling context management to reduce performance overhead. Sumner et al. [31] and Zeng et al. [32] proposed calling context encoding that statically analyzes the whole program and assigns a single integer number for each context. Unlike the previous profilers [28]–[30] and calling context encoding [31], [32], CAMP constructs a context tree for every function call site and loop invocation, and encodes the call sites and loop nests. Since the context of CAMP reflects not only call sites but also loop nests, CAMP additionally has an iteration stack to store iteration counts of each loop in a loop nest.

VII. Conclusion

This paper proposes a new precise compiler-runtime cooperative context-aware memory profiling (CAMP) framework that traces memory dependences with their full context information such as call site stacks and loop nest levels. For 14 programs from SPEC CINT2000 and CINT2006 benchmark suites, context-awareness increases speculative parallelism opportunities by 12.6% on average by finding that 87.6% of loop-aware memory profiling results are false positive. With compiler-assisted context management, CAMP suffers from only 47.2% and 28.0% of additional profiling time and memory usage while collecting 1.98× more dependences than loop-aware memory profiling.

Acknowledgment

We thank the anonymous reviewers for their insightful comments and suggestions. This work is supported by Korea Institute of Science and Technology Information (KISTI) under the grant No.K-16-L01-C03-S03, and the Korean Government (MSIT) under the "Next-Generation Information Computing Development Program" (NRF-2015M3C4A7065646), the "Basic Science Research Program" (NRF-2017R1C1B3009332), the "ICT Consilience Creative Program" (IITP-2017-R0346-16-1007), and the "PF Class Heterogeneous High Performance Computer Development" (NRF-2016M3C4A7952587).

References

[1] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August, "Revisiting the sequential programming model for multi-core," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[2] D. A. Connors, "Memory profiling for directing data speculative optimizations and scheduling," Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1997.

[3] N. P. Johnson, H. Kim, P. Prabhu, A. Zaks, and D. I. August, "Speculative separation for privatization and reductions," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[4] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas, "POSH: a TLS compiler that exploits program structure," in *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2006.

[5] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation," in *Proceedings of the 27th International Symposium on Computer Architecture*, 2000.

[6] W. Thies, V. Chandrasekhar, and S. Amarasinghe, "A practical approach to exploiting coarse-grained pipeline parallelism in c programs," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[7] P. Wu, A. Kejariwal, and C. Caşcaval, "Languages and compilers for parallel computing," 2008, ch. Compiler-Driven Dependence Profiling to Guide Program Parallelization.

[8] X. Zhang and S. Jagannathan, "Alchemist: A transparent dependence distance profiling infrastructure," in *Proceedings of the 2009 International Symposium on Code Generation and Optimization*, 2009.

[9] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke, "Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[10] C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen, "Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[11] L. Rauchwerger and D. A. Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization," *IEEE Transactions on Parallel Distributed Systems*, 1999.

[12] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry, "The STAMPede approach to thread-level speculation," *ACM Transactions on Computer Systems*, 2005.

[13] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke, "Uncovering hidden loop level parallelism in sequential applications," in *IEEE 14th International Symposium on High Performance Computer Architecture*, 2008.

[14] C. Zilles and G. Sohi, "Master/slave speculative parallelization," in *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, 2002.

[15] J. R. Larus, "Loop-level parallelism in numeric and symbolic programs," *IEEE Transactions on Parallel and Distributed Systems*, July 1993.

[16] A. Ketterlin and P. Clauss, "Profiling data-dependence to assist parallelization: Framework, scope, and optimization," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[17] M. Kim, H. Kim, and C.-K. Luk, "SD^3: A scalable approach to dynamic data-dependence profiling," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[18] R. Vanka and J. Tuck, "Efficient and accurate data dependence profiling using software signatures," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 2012.

[19] H. Yu and Z. Li, "Fast loop-level data dependence profiling," in *Proceedings of the 26th ACM International Conference on Supercomputing*, 2012.

[20] T. Chen, J. Lin, X. Dai, W.-C. Hsu, and P.-C. Yew, "Data dependence profiling for speculative optimizations," in *Compiler Construction*, 2004.

[21] Y. Sato, Y. Inoguchi, and T. Nakamura, "Whole program data dependence profiling to unveil parallel regions in the dynamic execution," in *IEEE International Symposium on Workload Characterization*, 2012.

[22] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[23] H. Kim, N. P. Johnson, J. W. Lee, S. A. Mahlke, and D. I. August, "Automatic speculative DOALL for clusters," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 2012.

[24] "Standard Performance Evaluation Corporation," http://www.spec.org, SPEC.

[25] H. Kim, A. Raman, F. Liu, J. W. Lee, and D. I. August, "Scalable speculative parallelization on commodity clusters," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[26] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August, "Speculative decoupled software pipelining," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[27] Y. Sato, Y. Inoguchi, and T. Nakamura, "On-the-fly detection of precise loop nests across procedures on a dynamic binary translation system," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, 2011.

[28] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, 1997.

[29] N. Froyd, J. Mellor-Crummey, and R. Fowler, "Low-overhead call path profiling of unmodified, optimized code," in *Proceedings of the 19th Annual International Conference on Supercomputing*, 2005.

[30] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi, "Accurate, efficient, and adaptive calling context profiling," in *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, 2006.

[31] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang, "Precise calling context encoding," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, 2010.

[32] Q. Zeng, J. Rhee, H. Zhang, N. Arora, G. Jiang, and P. Liu, "DeltaPath: Precise and scalable calling context encoding," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2014.