**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].
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[l]` is changing for L2, LD1@CS4@CS2 and ST1@CS5@CS2 of different iterations do not access the same value and 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.

---

**TABLE I**

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

---

1. int getValue(Node n) {
   2.   return n.value;  // LD1
3. }

4. void setValue(Node n, int v) {
   5.   if(isValid(v))  // CS7
   6.     n.value = v;  // ST1
   7. }

8. int work(Node n) {
   9.   int v1 = getValue(n);  // CS4
10.  int v2 = update(v1);  // CS5
11.  setValue(n, v2);  // CS6
12.  return v2;
13. }

14. void main() {
15.   for(int t = 0; t < T; t++) {
16.     for(int i = 0; i < N; i++) {
17.       int s = getValue(sum[t]);  // CS1
18.       int v = work(nodes[i]);  // CS2
19.       s = s + v;  // ADD
20.       setValue(sum[t], s);  // CS3
21.     }
22.   }
23. }

---

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

(b) Speculative PDG with loop-aware profiling results

---

Fig. 1. Example program

Fig. 2. A loop-aware profiling result and a simplified speculative program dependence graph of the example program in Figure 1.
of speculative parallel program. Figure 5 shows the performance impact of the context-aware profiling for speculative 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 geometric mean speedup of 8.1x, speculative parallelization with context-aware profiling results increases parallelism opportunities for 164.gzip, 197.parser, and 256.bzip2, and achieves a geometric mean speedup of 26.8x.

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 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
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
int getValue(Node *n) {  
    profiling_load(&(n->value), LD1);  
    return n->value;  // LD1
}

void setValue(Node *n, int v) {  
    change_context(+1);  
    bool t = isValid(v);  // CS7  
    change_context(-1);  
    if(t) {  
        profiling_store(&(n->value), ST1);  
        n->value = v;  // ST1
    }
}

int work(Node *n) {  
    change_context(+1);  
    int v1 = getValue(n);  // CS4  
    change_context(-1);  
    change_context(+2);  
    int v2 = update(v1);  // CS5  
    change_context(-2);  
    change_context(+3);  
    setValue(n, v2);  // CS6  
    change_context(-3);  
    return v2;
}

void main() {
    begin_loop(+1);  
    for(int t = 0; t < T; t++) {  // L1
        begin_loop(+1);  
        for(int i = 0; i < N; i++) {  // L2
            change_context(+1);  
            int s = getValue(sum[t]);  // CS1  
            change_context(-1);  
            change_context(+2);  
            int v = work(nodes[i]);  // CS2  
            change_context(-2);  
            s = s + v;  // ADD  
            change_context(+7);  
            setValue(sum[t], s);  // CS3  
            change_context(-7);  
            next_iteration();
        }
        end_loop(-1);  
        next_iteration();
    }
    end_loop(-1);
}

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(Ctx9) 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
Algorithm 1: Context-aware Dependence Generation

```
/* Update Dependence Table */
foreach (srcID, srcCtx, srcIterStack) \in getHistory(addr) do
  let dep = getDependence(srcID, srcCtx, dstID, dstCtx);  
  if dep == NULL then
    dep = createDep(srcID, srcCtx, dstID, dstCtx);
  end
  let depIter = getDependenceIter(dep);
  foreach level = 0 to minStackLevel(srcIterStack, dstIterStack) do
    if srcIterStack[level] == dstIterStack[level] then
      depIter[level] = INTRA;
    else
      depIter[level] = INTER;
    end
  end
end
/* Update History Table */
if dstID == STORE then
  replaceElement(addr, dstID, dstCtx, dstIterStack);
else
  clearLoadHistory(addr);
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 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 CAMP runtime 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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># of Static Instances</th>
<th># of Dynamic Instances</th>
<th># of P'll Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Functions</td>
<td>Loops</td>
<td>Call Sites</td>
</tr>
<tr>
<td>052.alvinn</td>
<td>9</td>
<td>39</td>
<td>2</td>
</tr>
<tr>
<td>164.gzip</td>
<td>70</td>
<td>200</td>
<td>462</td>
</tr>
<tr>
<td>175.xpr</td>
<td>155</td>
<td>482</td>
<td>2299</td>
</tr>
<tr>
<td>177.mesa</td>
<td>1019</td>
<td>1340</td>
<td>4827</td>
</tr>
<tr>
<td>179.art</td>
<td>26</td>
<td>132</td>
<td>274</td>
</tr>
<tr>
<td>197.parser</td>
<td>323</td>
<td>883</td>
<td>1066</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>74</td>
<td>233</td>
<td>239</td>
</tr>
<tr>
<td>300.twolf</td>
<td>190</td>
<td>1082</td>
<td>2294</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>69</td>
<td>301</td>
<td>487</td>
</tr>
<tr>
<td>433.milc</td>
<td>235</td>
<td>329</td>
<td>2680</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>467</td>
<td>1124</td>
<td>5168</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>95</td>
<td>119</td>
<td>568</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>948</td>
<td>2608</td>
<td>3521</td>
</tr>
<tr>
<td>470.lbm</td>
<td>19</td>
<td>44</td>
<td>87</td>
</tr>
</tbody>
</table>

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 \((\text{sum}[0].\text{value}|\text{LD}1|\text{Ctx}3:2/1)\), the runtime looks up the history table for the same address, and finds a store context \((\text{ST}1|\text{Ctx}9:1/1)\). 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 \textit{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 \textit{sum}[0].\textit{value} is \textit{0x00ACCE55}, the runtime allocates its corresponding history element at \textit{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 \textit{LD}1|\textit{Ctx}3:2/1, 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
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
