Master's Thesis

Context-Aware Memory Dependence Profiling

Juhyun Kim (김 주 현)

Department of Computer Science and Engineering

Pohang University of Science and Technology

2017

콘텍스트를 인지하는
메모리 의존성 프로파일링

Context-Aware Memory Dependence Profiling

Context-Aware Memory Dependence Profiling

by

Juhyun Kim

Department of Computer Science and Engineering

Pohang University of Science and Technology

A thesis submitted to the faculty of the Pohang University of Science and Technology in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science and Engineering

Pohang, Korea

December 1, 2017

Approved by

Prof. Hanjun Kim

Academic Advisor

Context-Aware Memory Dependence Profiling

Juhyun Kim

The undersigned have examined this thesis and hereby certify that it is worthy of acceptance for a Master's degree from POSTECH.

11/21/2017

Committee Chair	Hanjun Kim
Member	Jangwoo Kim
Member	Kyungmin Bae

MCSE 김주현 Juhyun Kim,

20152083 Context-Aware Memory Dependence Profiling
콘텍스트를 인지하는 메모리 의존성 프로파일링
Department of Computer Science and Engineering, 2017, 64P,
Advisor: Hanjun Kim
Text in English.

ABSTRACT

To support aggressive optimizations, many researchers employ data dependence profilers which identify dynamic dependence patterns in a program. Although their analysis motivates more beneficial PDGs (i.e. speculative PDGs), data dependence profilers that are not sensitive to the program contexts, such as function call sites and loop nest levels, are likely to produce false results. I propose a context-aware memory profiler (CAMP) which traces memory dependencies with their full context information. CAMP is a compiler-runtime cooperative system which takes advantage of a static analysis to ease the overheads of context management in profiling, without compromising precision, coverage, or performance of profiling. Preventing from generating lots of false dependencies, CAMP enables compilers to build context-aware speculative PDGs that are more precise than what a context-oblivious profiler makes. I show how a precise context-aware PDG facilitates a compiler optimization such as speculative parallelism. For 12 programs from SPEC benchmark suites, the evaluation results show that CAMP successfully removes significant number of false dependencies which take 70.8% of total dependencies that a context oblivious profiler makes. In the evaluation, CAMP finds false dependencies which take 73.3% of all possible memory dependencies at the finest granularity (i.e. instruction-pairwise and byte-level), while showing only 18.4× slowdown.

Contents

List of Figures

List of Tables

List of Equations

Chapter 1

Introduction

Dependence information is essential for many compiler-assisted optimizations. In order to correctly transform programs, a compiler performs various analyses to collect information of dependence in the program. Control dependence information tells which portion of the program depends on which branch, call, or jump instruction. Data dependence information tells sets of instructions which access or update the same memory address so that a compiler can preserve correct orders of instructions. As a preparatory analysis for compiler optimizations, these types of dependence information are consolidated into a *Program Dependence Graph (PDG)* which illustrates the overall program structure and behavior.

Although PDGs are widely used in many optimizations, they often fail to facilitate aggressive optimizations due to conservative static analyses on memory dependencies. Since compilers cannot determine the exact dependencies in programs, a statically constructed PDG gives the most conservative version of dependence information. Due to aliased pointers, compilers insert a great number of false dependencies into PDG, especially in languages that allow explicit use of pointer-based references. Limiting the

analyses that depend on PDG, these false dependencies adversely affect aggressive optimizations such as parallelization, offloading, and approximation.

To aggressively optimize programs, modern compilers [3, 5, 7, 13, 19, 21, 23, 25] employ memory profilers that trace dynamic memory dependences among instructions. Once rarely occurring dependences are identified via profiling, compilers speculatively remove the rarely occurring dependences from a program dependence graph (PDG) generating a speculative PDG. With the speculative PDG, the compilers can support aggressive optimization such as speculative parallelism. For example, even if independence among iterations cannot be proven statically, the compilers optimistically exploit loop-level parallelism when there is no inter-iteration dependence during profiling [13, 14, 15, 16, 20, 26, 28]. Therefore, generating a precise speculative PDG with high-coverage is crucial to enlarge aggressive optimization opportunities.

Context-aware representation is essential for PDG to clarify data dependencies. Even for the same instruction pairs, data dependence patterns vary widely depending on the program context, such as a function call site stack and a loop nest level. Without call site contexts, a PDG cannot distinguish data accesses of the same functions from different call sites. Then, a compiler will conservatively insert data dependencies into the PDG between all the call sites, generating lots of false dependencies. Moreover, in nested loops, there may exist inter-iteration dependencies between two instructions

in an inner loop, while not in an outer loop. If the PDG cannot distinguish two different loop contexts, the inter-iteration dependencies will be associated with both the inner loop and the outer loop. By excluding these false dependencies, a context-aware PDG gives more precise data dependence information.

Although many researchers have proposed context-aware memory profilers that trace dynamic memory dependencies with context information, their tools suffer from severe overheads in terms of CPU cycles and memory space, and tracing all memory dependences with their contexts easily become impractical. In general, profiling memory dependencies greatly increases instruction counts to identify and record dependencies between instructions that touch the same memory address. Context awareness exacerbates this problem by separately treating the same pairs of instructions whose contexts differ. The profiler of [10], for example, shows over 250 times slowdown (serial version) because of its significant costs of managing history table whose entries are associated with loop iterations.

Most existing memory profilers circumvent this problem at the expense of quality attributes of profiling. Targeting on a few specific optimizations, such as parallelization, [8, 10, 11, 22, 24] narrow down their scope of context information into loops, which harms availability of the profiling result to other optimization clients. Only a few memory profilers [4, 18] log full context information including function call sites as well as loops, albeit they compromise their precision by either using compacted context information or

representing dependencies in context granularity. Table 1.1 summarizes and compares the existing memory profilers.

System	Loop-Aware	Call Site-Aware	Full Coverage of Dependences	Whole Program Coverage	Profiling Granularity
H. Yu et al. [24]	✓	✗	✓	✗	Variable
A. Ketterlin et al. [8]	✓	✗	✓	✓	Variable
R. Vanka et al. [22]	✓	✗	✗	✓	Byte
M. Kim et al. [10]	✓	✗	✓	✓	Byte
T. Chen et al. [4]	✓	Compacted Call Path	✗	✓	Byte
Y. Sato et al. [18]	✓	✓	✗	✓	Context
CAMP [This thesis]	✓	✓	✓	✓	Byte

Table 1.1 Comparison of memory profiling systems

To generate precise context-aware PDGs, this paper proposes a new compiler-runtime cooperative Context-Aware Memory Profiler (CAMP) which traces memory dependencies in a byte level granularity. While preserving full context information, we focus on the instrumentation interface between static and dynamic analysis, so the former helps the latter to greatly reduce the overheads of managing contexts. The CAMP compiler statically generates a context tree which represents all the possible contexts in a PDG, and provides the CAMP runtime with static context offsets as hints to achieve

dynamic context IDs for every call site and loop. The CAMP runtime calculates a dynamic context ID with one arithmetic operation between the current context ID and the static offset, and records a memory access history with the context ID. Using one dynamic context ID from a simple operation simplifies the data structure and the algorithm of CAMP, and minimizes its profiling time and memory overheads.

To further reduce profiling time, this paper also proposes a new heterogeneous sampling method that does not generate any false positive dependencies. Since memory profilers collect dependences that *really* manifest for given profiling inputs, the full-profiling results do not include any false positive (there is no dependence, but reported as having one) while the results may include false negative (there is a dependence, but reported as none) for non-travelled control flows. However, sampling memory instructions may yield false positive dependences because the profilers may link memory reads to wrong memory writes due to absence of memory write history. To avoid generating any false positive, CAMP adopts different sampling policies for read and write instructions.

For 12 programs from SPEC CINT2000 and CINT2006 benchmark suites, CAMP finds that 70.8% of context oblivious profiling results are false positive on geometric average, and increases DOALL parallelism opportunities by average 9.7% more than loop-aware only memory profilers. Compared to CAMP without sampling, CAMP with the heterogeneous sampling reduces the

profiling time by 10.8× (from 197.0× to 18.4× and memory overhead by 40% (from 2.7GB to 1.6GB) on average without any false positive result.

In summary, the primary contributions of this paper are:

- A compiler-runtime cooperative precise context-aware memory profiling system with full contexts (called CAMP)
- A static context tree that represents all the possible dynamic contexts such as function call site stacks and loop nests
- A novel heterogeneous sampling method that does not generate any false positive dependence
- An in-depth evaluation of CAMP using 12 benchmarks from SPEC CINT2000 and CINT2006 benchmark suites

The rest of this paper is organized as follows. Chapter 2 gives the motivation of this research and challenging issues in context-aware dependence profiling. Chapter 3 describes CAMP compiler and the proposed compiler technique. Chapter 4 explains the CAMP runtime and its algorithm. Chapter 5 explains the optimization method such as a novel sampling method which does not generate any false positive dependency. Chapter 6 discusses the meaning and the potential application of speculative context-aware PDGs, giving a case study of discovering hidden parallelism. Chapter 7 shows the evaluation results, and Chapter 8 contains related work. Finally, Chapter 9 concludes the thesis.

Chapter 2

Motivation

Context-aware dependence profiling is an essential step in building context-aware PDG. Although a profiler gives information about only specific program executions, it realizes empirically predictable PDG, often called speculative PDG. Supporting precise and fine-grained representations, Context-awareness plays an important role not only in speculative PDGs, but also in dependence profiling.

This section first motivates context-aware memory profiling and false positive-free sampling. Section 2.1 shows an example of real life program and a part of PDG of it, giving an idea of how context information affects data dependencies between different call sites and loops. It also shows how the context-awareness increases the precision of a speculative PDG by comparing with a speculative PDG with context-oblivious memory profiling results. Section 2.2 illustrates why speculative PDGs of the same function can vary depending on contexts and how context-awareness increases optimization opportunities such as uncovering hidden parallelism.

2.1. Dependence Information Quality: An Example of a Real Life Program

Compared with a naive context-oblivious profiler, a context-aware profiler provides concise data dependence information in terms of both quantity and quality. As an example of a real-life program, Figure 2.2 illustrates parts of memory dependencies that context-aware and context-oblivious memory profilers collect for a global variable `outcnt` in `gzip` (Figure 2.1). True dependencies, observed by a context-aware memory profiler, are drawn in blue lines, whereas false dependencies which occur only from context-oblivious memory profiling are drawn in red lines. Notice that every pair of `send_bits` function calls is correlated by both dependencies.

Compilers will conclude that any `send_bits` invocation is unable to reordered or parallelized due to false dependencies. These false dependencies originate from compiler's conservative assumptions on a single context-oblivious dependency, i.e. the compiler inserts dependencies on `outcnt++` for all the possible combinations of `send_bits` function calls. Given context-aware dependencies, the compiler can distinguish call sites of memory accesses, and prevent from generating false dependencies. Only for the variable `outcnt` in the whole program, the context-oblivious memory profiler claims 1,344,238 false dependencies, while there exist only 499 true memory dependencies. In addition, context-aware profiler elaborates the dependencies

by classifying it into intra-iteration (solid line) and inter-iteration (dotted line) with respect to a corresponding loop.

```
void send_bits(value, length) {
  if (bi_valid> (int)size -length) {
    bi_buf|= (value << bi_valid);
    outbuf[outcnt++] = bi_buf;
    bi_buf= (ush)value >> (size -bi_valid);
    bi_valid+= length -size;
  } else {
    bi_buf|= value << bi_valid;
    bi_valid+= length;
  }
}

void compress_block(ltree, dtree) {
  while(lx < last_lit} {
    flag = flag_update();
    lc= l_buf[lx++];
    if (flag == 0) {
      send_bits(lc, ltree); // 1
    } else {
      code = length_code[lc];
      send_bits(code, ltree); // 2
      extra = get_extra(code);
      if (extra != 0) {
        lc= get_lc(code);
        send_bits(lc, extra); // 3
      }
      code = d_code(dist);
      send_bits(code, dtree); // 4
      extra = get_extra(code);
      if (extra != 0) {
        dist= get_dist(code);
        send_bits(dist, extra); // 5
      }
    }
  }
  send_bits(END_BLOCK, ltree); // 6
}
```

Figure 2.1 Pseudo code of `compress_block` in `gzip`

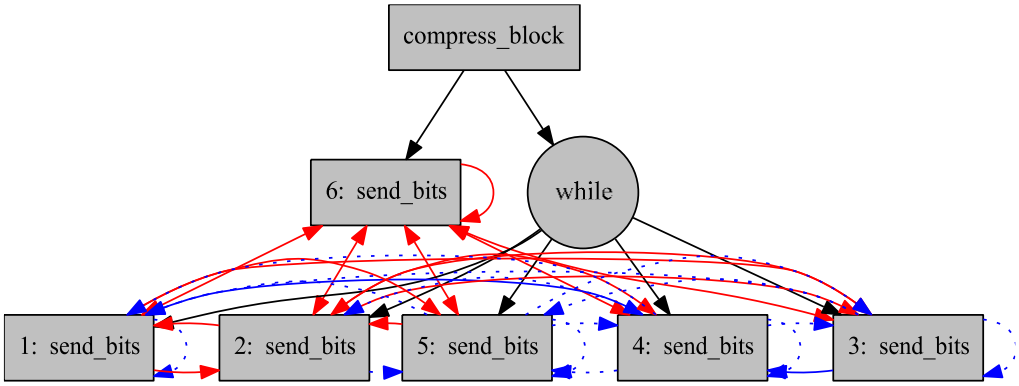


Figure 2.2 A part of Speculative Program Dependence Graph of Figure 2.1

2.2. Recording Context Information

A precise speculative PDG with memory profiling results about dynamic dependences enables modern compilers [3, 5, 7, 13, 19, 21, 23, 25] to support aggressive optimization that cannot be achieved by static analyses only. For example, automatic speculative parallelizing compilers [9, 13, 14, 15, 16, 20, 26, 28] collect dynamic dependences in loops, and speculatively parallelize the loops ignoring rarely occurring dependences that static analysis cannot remove. Moreover, speculative PDGs help parallelizing compilers produce robust codes by augmenting fragile static analyses [9].

To draw context-aware dependencies, a profiler should incorporate context information into records of all memory access events. The dynamic dependences vary depending on their contexts such as loop nest levels and

function call stacks. Figure2.3 is a simple program which calculates moving weighted averages for matrices, and Figure 2.4 shows how context information refines and distinguishes a data dependency between a pair of instructions. Assuming that `tmp1` in function `wgtAvgV` is held in a register (which is highly likely), a Read-After-Write (RAW) memory dependency from `LD4` to `ST6` occurs only when `v1` and `v3` are aliased[1]. For example, `ST6` updates `A[t][i]` that `LD4` reads at the next iteration of Loop `L2` in Function call site `F3` (Notice that `wgtMovingAvg(A,A)` makes `v1` and `v2` aliased). Otherwise, the RAW dependency does not exist. As shown in Figure 2.4.3, the most verbose context information, which considers both loops and call sites, provides the finest resolution on dependencies. In this regard, the context information on memory access histories plays a key role in indicating an exact position of dependencies in speculative context-aware PDG.

A context-oblivious memory profiler that does not track calling context or loop nest severely limits the applicability of speculative parallelization. For example, such a profiler may report that there is a memory dependence from `ST6` to `LD4` with no available context information. As Figure 2.4.1 illustrates, since `LD4` and `ST6` form a cyclic dependence graph with the profiled memory dependence on `A[t][i]` and a statically found data dependence on `tmp1`,

[1] Since register dependencies manifest themselves at compile time, we exclude them from the scope of this paper.

automatic speculative parallelizing compilers are unable to parallelize both `Loop L1` and `Loop L2`.

Like most existing memory profilers [8, 19, 11, 22, 24], if a memory profiler is aware of loop nest levels, applicability of the speculative parallelization can be increased. As Figure 2.4.2 shows, the loop-aware memory profiler reports that there is an inter-iteration memory dependence from `ST6` to `LD4` for `Loop L2` but not for `Loop L1` because `LD4` reads `A[t-1][i]` that `ST6` updates at the previous invocation of `Loop L1`. Since there is no cyclic dependence in a dependence graph for `Loop L1`, the parallelizing compilers can parallelize `Loop L1` while the compilers still cannot parallelize `Loop L2`. Therefore, not to lose this parallelism opportunity on `Loop L1`, the memory profiler should record dependences with loop nest levels.

For more aggressive optimization, context information like function call site stacks is necessary. `Instruction 22` and `Instruction 24` call the same function `wgtMovingAvg` with different arguments such as (`A`, `B`) and (`A`, `A`), so `wgtMovingAvg` has different dependence graphs for each call site. As Figure 2.4.3 shows, a context-aware memory profiler records memory dependences for each context including call stacks, and allows compilers to generate different dependence graphs for loop nest levels and call stacks. As a result, the compilers can parallelize the outer loop `L2` for call site `F2` that cannot be parallelized with loop-aware memory profiling only. Moreover, context-awareness of memory profiling allows compilers to recognize memory

operations through getter and setter functions from different call sites as different memory operations, and to more aggressively optimize programs with mutator functions.

Augmenting context information, however, incurs significant overheads. Attaching heavy data such as function call stacks, loop nesting information, and iteration counters to each memory access history is the simplest but almost infeasible approach. The profiler of [10], instead, utilizes pending and history tables to trace context changes (loop iteration) of memory accesses within nested loops. Still, since these tables are managed per loop nest, they cause significant memory overheads when the loop nests are deep. In [8], the execution tree tracks and maintains context changes at profiling time. Besides managing the tree, their profiler needs to compute the latest common execution point of two memory accesses of a dependency, which also costs significant computational overheads. Consequently, most context-aware profilers suffer from serious computational and memory overheads, resulting in unacceptable profiling times.

```
1  void wgtAvgV(float *v1, float *v2, float *v3) {
2    // Loop L1
3    for(int i = 0; i < N ; i++){
4      float tmp1 = v1[i];                  // LD4
5      float tmp2 = v2[i];
6      v3[i] = w * tmp1 + (1-w) * tmp2; // ST6
7    }
8  }
9
10 void wgtMovingAvg(float **in, float **out) {
11   // Loop L2
12   for(int t = 1; t < N ; t++){
13     // Function Call Site F1
14     wgtAvgV(in[t-1], in[t], out[t]);
15   }
16 }
17
18 void main() {
19   float A[N][N], B[N][N];
20   float v[N], w[N];
21   // Function Call Site F2
22   wgtMovingAvg(A, B);
23   // Function Call Site F3
24   wgtMovingAvg(A, A);
25   // Function Call Site F4
26   wgtAvgV(v, w, v);
27 }
```

Figure 2.3 Example Program

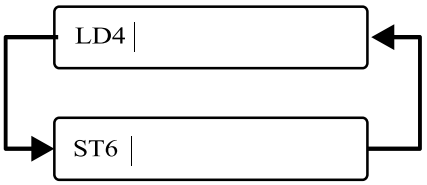


Figure 2.4.1 Context Oblivious (Loop and Call Site Oblivious)

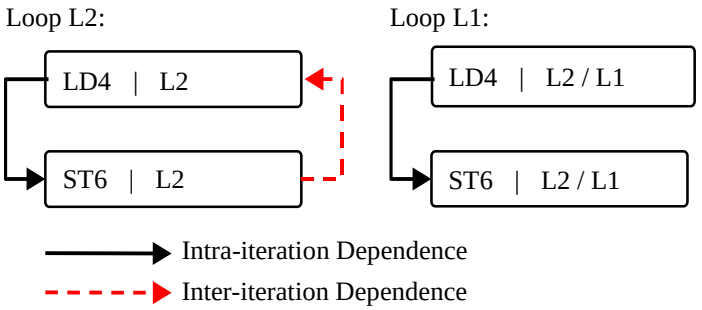


Figure 2.4.2 Loop Aware (Call Site Oblivious)

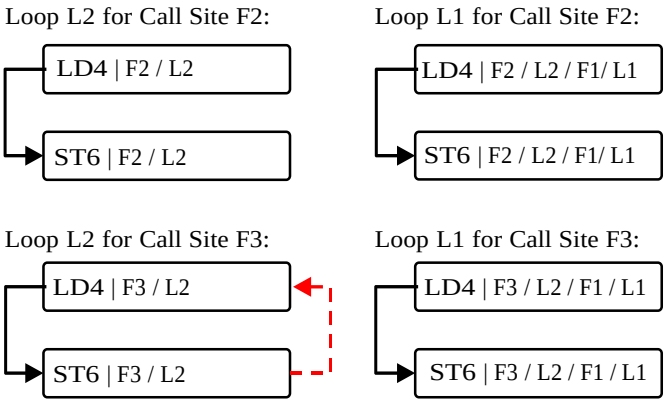


Figure 2.4.3 Context Aware (Loop and Call Site Aware)

2.3. Observations: Predictable Aspects of Contexts

To address the aforementioned problems, compilers can help profilers to reduce the cost of dealing context in several ways. A compiler could remove the need for managing context information by in-lining all function calls and unrolling loops, unifying all contexts into one; this radical approach seems plausible, but the size of program code will explode if the program has deep loop nests or function invocations. As another way to alleviate context management costs, compilers can provide concise representations for profilers to efficiently handle contexts at profiling time; since a compiler statically analyzes a target program before inserting instrumentation functions, it can give hints on how a profiler efficiently represents and computes contexts by leveraging its prior knowledges of the program.

Based on compiler assistance, we make several observations regarding predictable aspects of contexts. First, given full access to the source code, we can statically enumerate every possible context of a program by analyzing control flow in context granularity. In other words, we can statically construct a context tree such as Calling Context Tree (CCT) [2, 6, 27] and Loop Call Context Tree (LCCT) [17, 18], and assign each node a unique context ID. Second, by using a simple identification methodology on context ID, profilers can efficiently track these IDs even without carrying the context tree. Third, as [10] also pointed out, dependencies between iterations often exhibit stride patterns in both a loop and loop nests. To avoid wasteful duplications of

context information regarding loops, profilers only need to tell whether a dependency is inter-iteration or intra-iteration, which is sufficient for both context-aware PDGs and various code optimization clients.

Chapter 3

Compiler-assisted Context Management

Though Chapter 2 points out the necessity of context-aware memory profiling, profiling memory dependences with full contexts suffers from huge profiling time overheads. To overcome the high profiling overheads, the CAMP compiler statically analyzes a program before profiling and simplifies context management of the CAMP runtime.

3.1. Static Context Tree

To efficiently manage the context changes, we proposes a new static context tree that represents all the possible contexts during the program execution. Unlike the existing context trees such as Loop Call Context Tree (LCCT) [17, 18] and Calling Context Tree (CCT) [2, 6, 27] that profilers dynamically generate at profiling time, the CAMP compiler statically generates the context tree to alleviate context management overheads at profiling time.

The CAMP compiler creates the context tree in two steps; context tree generation and context ID assignment. At the context tree generation step, the compiler creates a context tree by inserting a child node for every function call site and loop invocation. Figure 3.1 shows a context tree for the example code

in Figure 2.3. The compiler inserts three different children into the `main` node for call sites `F2`, `F3` and `F4`. Here, though call sites `F2` and `F3` call the same function `wgtMovingAvg`, the compiler inserts different context nodes for each call site, thus generating different context nodes for the same call site `F1` and loop invocations `L1` and `L2` in `wgtMovingAvg`. As a result, the context tree can differentiate memory instructions across all the dynamic contexts.

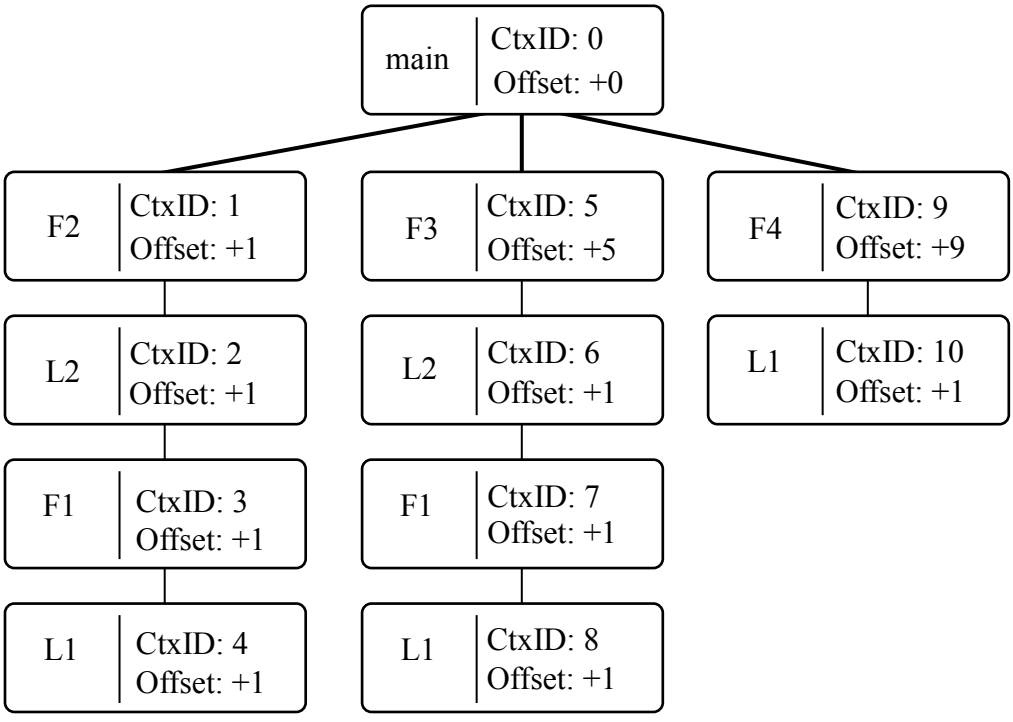


Figure 3.1 Context tree for the example code in Figure 2.3

At the context ID assignment step, the CAMP compiler assigns a unique context ID to each context node. Since there can exist multiple context nodes for the same call site and loop invocation such as `F1`, `L1` and `L2` depending on the dynamically determined context stack, the compiler needs to assign different context IDs at the same static call site and loop invocation codes. Addressing the problem, the compiler assigns context IDs as a unique path sum where each call site and loop invocation has the same static offset, and calculates the context IDs and their static offset with pre-order tree traversal. Figure 3.1 shows context IDs and their static offsets for the context tree. Though `L1` is invoked in multiple contexts with different IDs such as `4`, `8` and `10`, its static offset from its parent contexts is one value, `+1`. The CAMP profiler 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. In contrast to dynamically created context trees, static context trees are ignorant of recursion depth and the target function that a function pointer points to. So, the CAMP compiler marks recursive functions (including mutually recursive functions) before generating the context tree, and inserts only the first recursive function call site as a leaf *loop* context node, preventing from generating a context tree infinitely. 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 target candidates and inserts the candidates as children nodes.[2] Figure 3.2.1 shows an example code with recursive function calls and indirect function calls. Figure 3.2.2 illustrates that the compiler adds call site `F3` 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 `F4`.

[2] Given full access to the source code, the CAMP compiler determines the candidates by matching type signatures of all functions.

```
bool is_even(unsigned int n) {
  if (n==0) return true;
  else return is_odd(n-1);  // Call Site F1
}
bool is_odd(unsigned int n) {
  if (n==0) return false;
  else return is_even(n-1); // Call Site F2
}
```

```
int inc(int n) {
  return n+1;
}
int dec(int n) {
  return n-1;
}
```

```
void main() {
  int (*fPtr)(int);
  if(is_even(n)) // Call Site F3
    fPtr = &inc;
  else
    fPtr = &dec;
  fPtr(n);      // Call Site F4
}
```

Figure 3.2.1 Recursive/indirect function call example

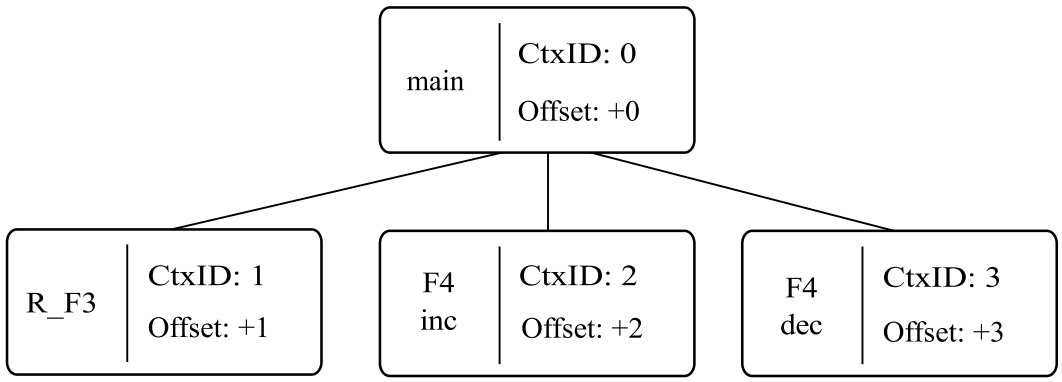


Figure 3.2.2 Context tree for Figure 3.2.1

3.2. Context Management and Profiling Code Generation

To track context changes, CAMP instruments not only memory accesses but also function calls and loops. During the program execution, contexts such as function 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 3.3 shows how the CAMP compiler inserts the notifiers to the example in Figure 2.3.

```
1  void wgtAvgV(float *v1, float *v2, float *v3) {
2    begin_loop(+1);
3    for(int i = 0; i < N ; i++){
4      profiling_load(&v1[i]);
5      float tmp1 = v1[i];
6      profiling_load(&v2[i]);
7      float tmp2 = v2[i];
8      profiling_store(&v3[i]);
9      v3[i] = w * tmp1 + (1-w) * tmp2;
10     next_iteration();
11   }
12   end_loop(-1);
13 }
14
```

```
15 void wgtMovingAvg(float **in, float **out) {
16   begin_loop(+1);
17   for(int t = 1; t < N ; t++){
18     profiling_load(&in[t-1]);
19     profiling_load(&in[t]);
20     profiling_load(&out[t]);
21     begin_function(+1);
22     wgtAvgV(in[t-1], in[t], out[t]);
23     end_function(-1);
24     next_iteration();
25   }
26   end_loop(-1);
27 }
28
29 void main() {
30   float A[N][N], B[N][N];
31   float v[N], w[N];
32   begin_function(+1);
33   wgtMovingAvg(A, B);
34   end_function(-1);
35   begin_function(+5);
36   wgtMovingAvg(A, A);
37   end_function(-5);
38   begin_function(+9);
39   wgtAvgV(v, w, v);
40   end_function(-9);
41 }
```

Figure 3.3 Transformed program by the CAMP compiler for the program in Figure 2.3. Bold lines are added by the CAMP compiler.

- **begin_function/begin_loop(offset)** notifies the beginning of a new context to the CAMP profiler with offset. The profiler calculates the new

context ID by adding the offset to the current context ID. If a loop context begins, the profiler pushes an iteration counter to the iteration counter stack that has iteration counts of loop nests.

- **end_function/begin_loop(offset)** notifies the end of the current context to the profiler with offset. The profiler restores the previous context ID by adding the offset to the current context ID. If a loop context ends, the profiler pops the iteration counter from the iteration counter stack.

- **next_iteration()** notifies the iteration change to the CAMP profiler. Since only the loop context at top of the stack (i.e., the inner most loop) can iterate, there is no argument in this mark. The profiler increases the iteration counter at the top.

With the context changing notifiers, the CAMP runtime efficiently 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 and a loop nest, the context manager adds (subtract) the

context offset into (from) the current context ID. To manage dependencies in the iterated contexts, CAMP has a global iteration counter stack which keeps how many times each context in the context stack iterates. For every loop iteration, CAMP changes the iteration information in the corresponding iteration counter.

Chapter 4

Context-Aware Memory Profiling

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

4.1. Overall Algorithm of Context-Aware Memory Profiling

CAMP runtime incorporates a context into every single dependency. 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 dependency, it needs to merge the instruction context into a dependence context that represents the context where the dependency is valid. For example, an inter-iteration RAW dependency in Figure 2.4.3 is valid at Loop `L2` invoked by `F3`, but is not valid at Loop `L2` invoked by `F2` nor Loop `L3`. Therefore, when adding the inter-iteration

dependency, CAMP should record its valid context such as Loop `L2` invoked by `F3`.

Figure 4.1 describes how the CAMP runtime generates dependencies with valid contexts. When an instruction accesses a memory location, the runtime receives the memory address and instruction ID, and has the current context ID and iteration counts of nested loops. First, the CAMP runtime searches previous memory instructions that access the same memory address from its history table (Line 1), and generates unique dependence IDs from the instruction IDs and context IDs of the current instruction and all the previous instructions (Line 2).Since the dependence ID reflects instructions and their contexts, CAMP can differentiate dependencies with contexts. Moreover, the dependence ID enables us to infer its dependence type such as RAR, RAW, WAR and WAW because the instruction IDs involve their instruction types such as load and store. Then, the CAMP runtime calculates iterative relation of the dependency by comparing each iteration count in the iteration stacks (Line 4-10). 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. To avoid inserting redundant dependencies, the runtime inspects the existence of the same dependencies in the dependence table (Line 11-20). If there exists the same dependencies with the same context, the runtime merges the iterative relations of the current and the existing dependencies. For example, if one dependency has inter-iteration relation and the other one has intra-iteration relation, the CAMP runtime marks

```
    Data: addr: accessed address
    Data: dstID: accessed instruction ID
    Data: dstCtx: current context ID
    Data: dstIterStack: current iteration stack
    /*  Generate Dependences                                    */
1   foreach (srcID, srcCtx, srcIterStack) ∈ getHistory(addr) do
2   |   let depID = genDependenceID(srcID, srcCtx, dstID, dstCtx);
3   |   let deptIter = NULL;
4   |   foreach level = 0 to minStackLevel(srcIterStack, dstIterStack) do
5   |   |   if srcIterStack[level] == dstIterStack[level] then
6   |   |   |   depIter[level] = INTRA;
7   |   |   else
8   |   |   |   depIter[level] = INTER;
9   |   |   |   break;
10  |   |   end
11  |   end
12  |   let dep = genDependence(depID);
13  |   if dep == NULL then
14  |   |   insertDependence(depID, depIter);
15  |   else
16  |   |   let oldDepIter = getDependenceIter(dep);
17  |   |   foreach level = 0 to maxStackLevel(oldDepIter, depIter) do
18  |   |   |   depIter[level] = oldDepIter[level] | depIter[level] ;
19  |   |   end
20  |   |   updateDependence(depID, depIter);
21  |   end
22  end
    /*  Update History Tables                                   */
23  if dstID == STORE then
24  |   replaceHistoryElement(addr, dstID, dstCtx, dstIterStack);
25  |   clearLoadHistory(addr);
26  else
27  |   addHistoryElement(addr, dstID, dstCtx, dstIterStack);
28  end
```

Figure 4.1 Context-aware Dependence Generation Algorithm

the dependency as `mixed`.

After generating the dependency, the CAMP runtime updates the history table with the new instruction. (Line 22-28). If the current instruction is a load, the runtime simply adds the current instruction and its context in the load history table. Whereas, 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 a WAR dependency is the relation between the current store and all the previous loads after the last store instruction.

4.2. 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 4.2(1) shows how the CAMP runtime creates memory instruction context from the memory event and the current context using the example code

in Figure 3.3 and the context tree in Figure 3.1. When `Instruction 4` accesses `A[1][0]` at `L1` called by `F1`, `L2` and `F3`, a memory event is notified with a memory instruction (`LD4`) and its memory address (`A[1][0]`). The runtime merges the instruction with context ID (`Ctx8`) and iteration counters (`2/1`), and generates a memory instruction context as `4:Ctx8(2/1)`. The generated instruction context will be used in the history table and dependence generation.

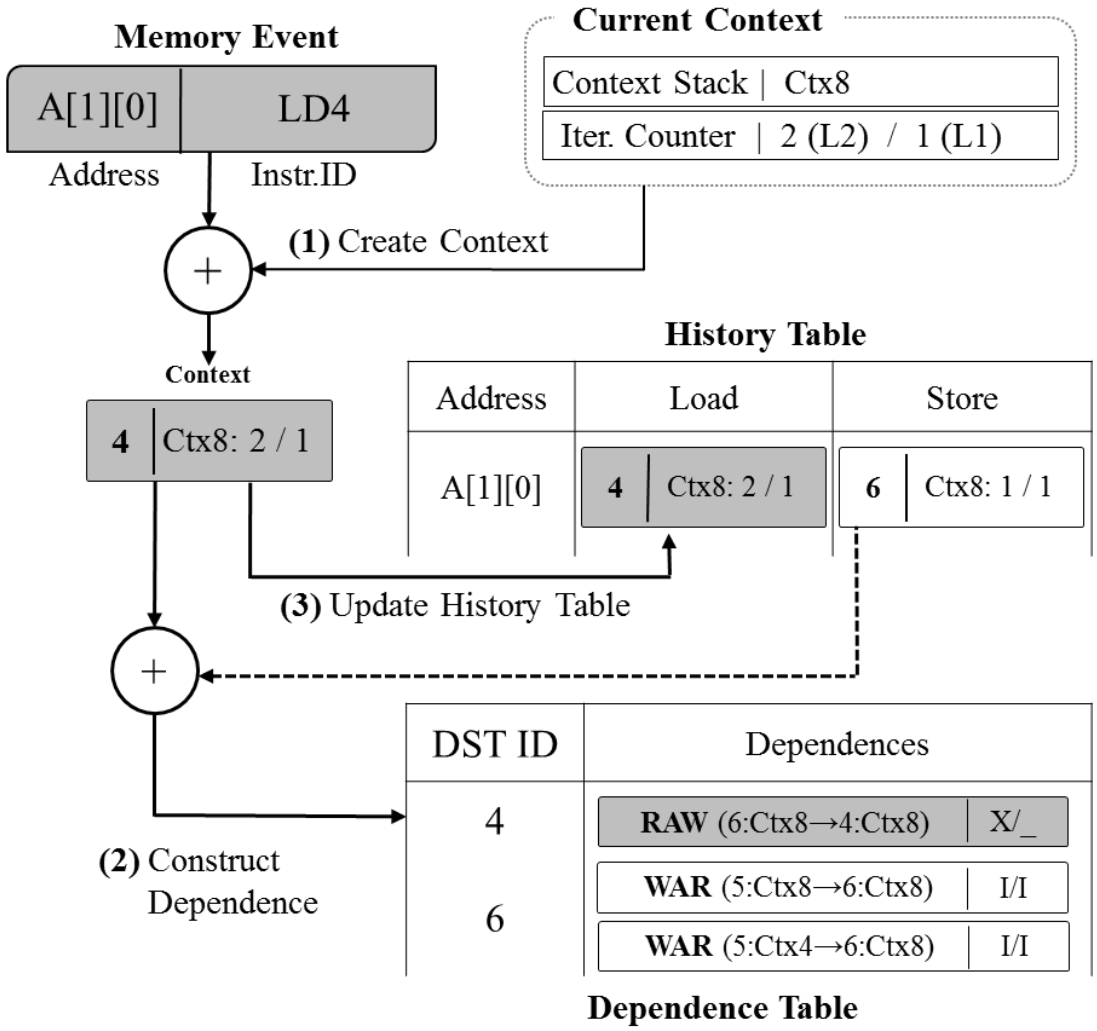


Figure 4.2 Structure of the CAMP runtime and its operation example on the program in Figure 3.3.[3]

[3] In the context, the number in left-most position means instruction ID, and the numbers after the context ID (Ctx8) are iteration counts of nested loops. In the dependence table element, the right half indicates loop iteration relation (I and X mean 'INTRA' and 'INTER', respectively). Updated elements by the operations in the figure are shaded in grey.

4.3. Dependence Table

While executing programs, the CAMP runtime directly generates RAW, WAR and WAW dependencies and records the dependencies 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 dependencies can share the same destination instruction, multiple dependencies can be stored in each element in the dependence table, so the runtime uses linked lists for each destination.

Figure 4.2(2) shows how the CAMP runtime generates a RAW dependency and stores the dependency in the dependence table from the example code in Figure 3.3. Given the instruction context (`4:Ctx8(2/1)`) and memory address (`A[1][0]`), the runtime looks up the history table for the same address, and finds a store context (`6:Ctx8(1/1)`). With the two instruction contexts, the runtime generates a context-aware dependency according to algorithm in Figure 4.1. Since the iteration counts are different at `L2`, the iterative relation is valid up to `L2`, and `L2` is marked as an inter-iteration dependence. Here, the CAMP runtime can safely ignore `F1` and `L1` contexts because the dependency exists only across different invocations for `F1` and `L1` and does not affect instruction reordering in `wgtAvgV` and `Loop L1`.

4.4. 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 dependencies between the current instruction and all the previous instructions in the history tables, and updates the history tables with the current instruction.

Figure 4.2(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 load like `4:Ctx8(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 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 WAR dependences between the current store instruction and a load instruction before the previous store instruction.

The history tables are implemented in *shadow memory*. It is logically orthogonal to original application address space. By mirroring the application address space, it enables the runtime to efficiently record and retrieve memory

access histories. When a memory address is accessed, the runtime calculates the corresponding *shadow address* simply with a few bit operations. A history data like `4:Ctx8(2/1)` can be found at these *shadow addresses*. In an on-demand fashion, the *shadow memory* is reactively allocated and freed at a page granularity, sparing lots of memory space. This idea of *shadow memory* is similar to the ones in [4, 8].

Chapter 5

Heterogeneous Sampling in CAMP

Memory profilers are very sensitive to false positives that affect compiler optimization. Since memory profilers collect dependences only for given profiling inputs, profiling results always involve false negative for non-travelled control flows. Thus, when compilers aggressively optimize programs with the profiling results, the compilers assume that false negative dependences can manifest at run-time. However, since the memory profilers collect dependences that really manifest, profiling does not generate any false positive dependence ideally, and compilers do not assume a false positive dependence in their optimization. Therefore, profiling optimization that can introduce false positive results may significantly affect compiler analysis and optimization.

To further optimize our profiling method, we propose a heterogeneous sampling method which employs two different sampling patterns together; *random sampling* and *consecutive sampling*. Since major overheads of profiling are associated with loops, we apply these two sampling methods only inside loops. *Random sampling* is to randomly choose loop iterations where all memory instructions are instrumented. Whenever CAMP encounters a new

iteration (i.e. `next_iteration()`), it randomly decides whether the iteration should be inspected, according to the predetermined random sampling ratio. *Consecutive sampling*, whereas, takes into account first several iterations of a loop to be instrumented. Dependence patterns in loops are usually straightforward (they often occurs consecutively, or in stride patterns.), so they are detectable in first several iterations. By taking advantage of this property, *consecutive sampling* catches most of regular dependencies in loops at an early stage, easing the burdens of *random sampling* later. To avoid exhaustive loop profiling, we apply these two different sampling method together in choosing which iterations to be inspected, yet finding most of dependencies in loops.

To avoid generating false positive dependencies, CAMP applies different sampling policies to memory reads and writes. Figure 5.1 shows how a careless sampling policy introduces false positive dependencies. Notice that only sampling memory write instructions introduces false positive because the absence of up-to-date memory write history can make a memory profiler generate a dependency with a wrong memory write instruction. For example, the absence of `ST2` leaves the memory write history on `Address A` not updated. Thus, the profiler generates dependencies with `ST1` instead of `ST2` for following memory instructions, and newly introduces false positive such as `WAW(ST1->ST3)` and `RAW(ST1->LD2)` that are red lines in Figure 5.1(c). To prevent this situation, CAMP updates history tables for all the memory writes. In other words, for non-sampled memory reads, it skips all the

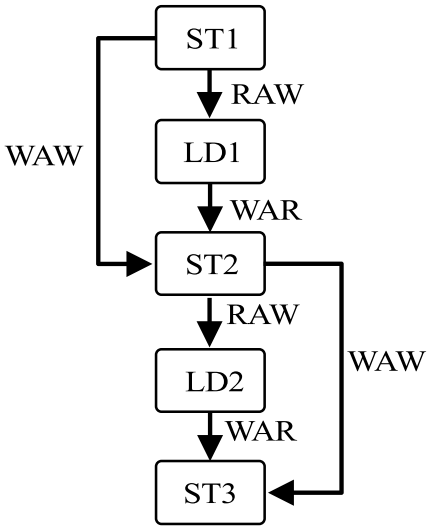
three steps of the routine in Figure 4.2. Whereas, for non-sampled memory writes, it skips only the dependence table update step. Since the number of memory writes is smaller than the number of reads, and updating the history tables is cheaper than updating dependence tables, CAMP prevents any false positive dependency without sacrificing much of the performance. Table 5.1 summarizes CAMP sampling policies.

Operations	Sampled		Not Sampled	
	Read	Write	Read	Write
Context Creation	✓	✓	×	✓
History Table Update	✓	✓	×	✓
Dependence Table Update	✓	✓	×	×

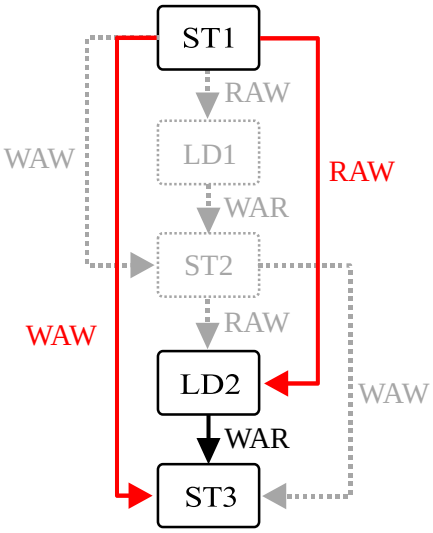
Table 5.1 Heterogeneous sampling for read and write

```
store A; //ST1, sampled
load A; //LD1, not sampled
store A; //ST2, not sampled
load A; //LD2, sampled
store A; //ST3,sampled
```

(a) Sequence of memory instructions



(b) Full profiling result



(c) Sampled profiling result

Figure 5.1 Dependences from full profiling and sampled profiling.[4]

[4] Grey means false negative, and red means false positive

Chapter 6

Context-Aware PDGs and Optimization Opportunities

By excluding a large number of false dependencies, CAMP helps a compiler to generate much more concise context-aware PDGs than a context-obvious profiler. Since a program often manipulates memory values through accessor functions such as getter and setter functions across all the program points, context-oblivious profiling forces compilers to conservatively insert dependencies into a context-aware PDG for all the combinations between getters and setters, spawning lots of false dependencies. Preventing such faults, CAMP allows compilers to distinguish memory accesses at different call sites. Figure 6.1 shows the ratios of false dependencies that CAMP finds from context oblivious memory profiling results. Here, the false dependencies are exactly the same concept of the red edges in Figure 2.2. We find that 70.8% of context oblivious memory profiling results for 12 programs are false.

In order to show their potential of context-aware speculative PDGs, this work performs a case study for an aggressive optimization, namely *speculative*

parallelism. Even if independence among iterations cannot be proven statically, this technique optimistically exploits loop-level parallelism when no inter-iteration dependency is exposed during profiling. It serves as a foundational technology for many automatic parallelizing compilers. More details are in [9, 13, 14, 15, 16, 20, 26, 28]. To see how much CAMP increases parallelism opportunities, we compare two numbers of parallelizable loops in the programs; one is estimated by CAMP and the other is estimated by a loop-aware only memory profiler (LAMP). A loop is considered as a parallelizable loop if the loop does not have any inter-iteration control, register and memory dependency except on induction variables. For simplicity, we only consider DOALL parallelism in this case study. Here, the LAMP profiler is equivalent to a CAMP profiler without function call-site awareness.

Figure 6.2 shows the increment of parallelizable loops by CAMP against LAMP. Compared with LAMP, CAMP increases parallelizable loops by 9.7% on average and by up to 54.2% (`401.bzip2`). As CAMP provides more concise dependence information, our compiler finds additional loops that is free of inter-iteration dependencies in most cases. These additional loops were considered to be not parallelizable by LAMP because, for example, functions that touches memory variables are invoked inside their loop body, or a single instance of them actually has an inter-iteration dependency in a certain context. Thanks to context-aware PDG, our compiler corrects these misjudgments by accepting more diversified contexts of a loop. CAMP, however, fails to increase parallelism opportunities for `177.mesa` and `462.libquantum`

because the programs have regular memory access patterns for LAMP enough to find parallelizable loops. CAMP also fails to increase parallelism for `188.ammp` and `429.mcf` because CAMP creates context trees with a small number of context nodes due to recursive calls.

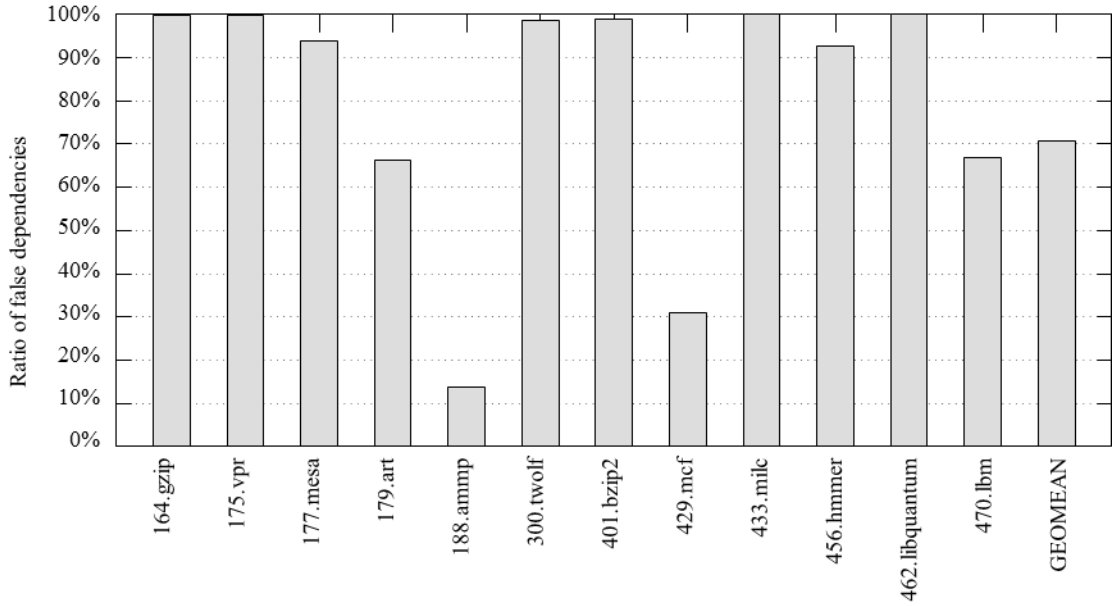


Figure 6.1 Ratio of false dependencies that CAMP finds from context oblivious memory profiling results

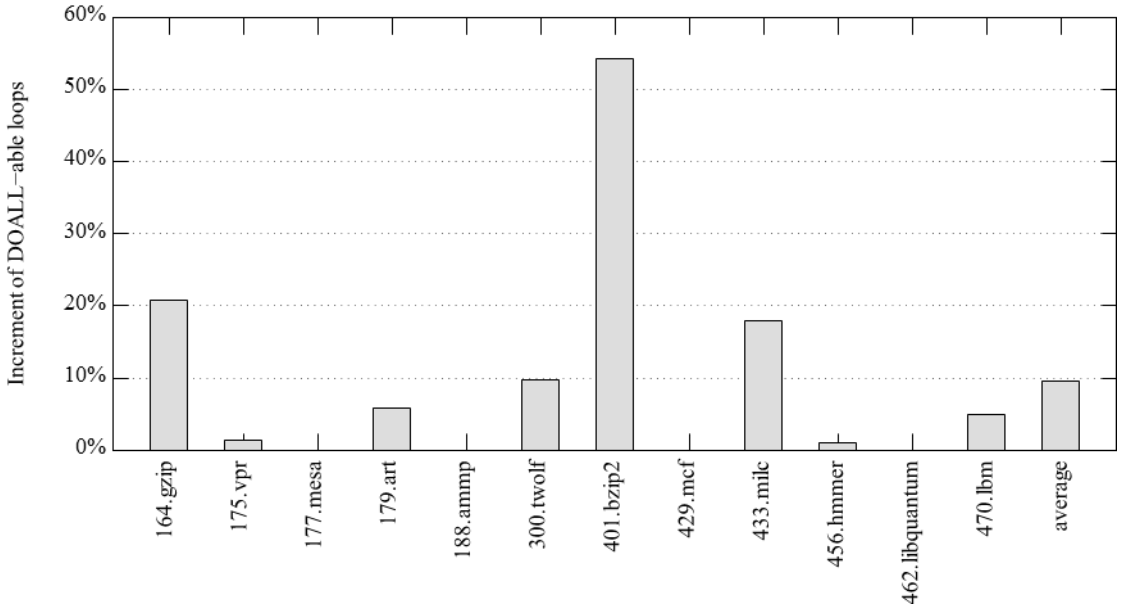


Figure 6.2 Increment of DOALL parallelizable loops with CAMP compared to LAMP

Chapter 7

Performance and Sampling Accuracy

We implemented the CAMP compiler and runtime on top of the LLVM compiler infrastructure [12] (revision 242,220). It is evaluated with 12 general-purposed programs in the SPEC CINT2000 and CINT2006 benchmark suites [1]. 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 7.1 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 [1]. The numbers of loops and call sites in the programs range from 153 (`429.mcf`) to 6,292 (`464.h264ref`), and the numbers of executed memory instructions also range from 101 million (`164.gzip`) to 13 billion (`179.art`).

Benchmark	# of Static Instances					# of Dynamic Instances			
	Functions	Loops	Call Sites	Loads	Stores	Calls	Loop Invo.	Loads	Stores
164.gzip	70	200	462	1191	1134	5M	1M	69M	32M
175.vpr	155	482	2299	4250	1336	113M	50M	2118M	573M
177.mesa	1019	1340	4827	16594	11744	3913M	8M	5356M	3751M
179.art	26	132	274	674	282	71M	255M	9810M	3359M
188.ammp	179	461	1453	4031	1336	183M	95M	6618M	1680M
300.twolf	190	1082	2294	10585	3773	12M	20M	407M	125M
401.bzip2	69	301	487	2514	1662	41M	101M	1116M	267M
429.mcf	24	58	95	372	292	3M	81M	1198M	138M
433.milc	235	329	2680	3498	1064	47M	7M	1505M	439M
456.hmmer	467	1124	5168	9739	4594	64M	47M	3316M	1853M
462.libquantum	95	119	568	646	345	182M	77M	5366M	2089M

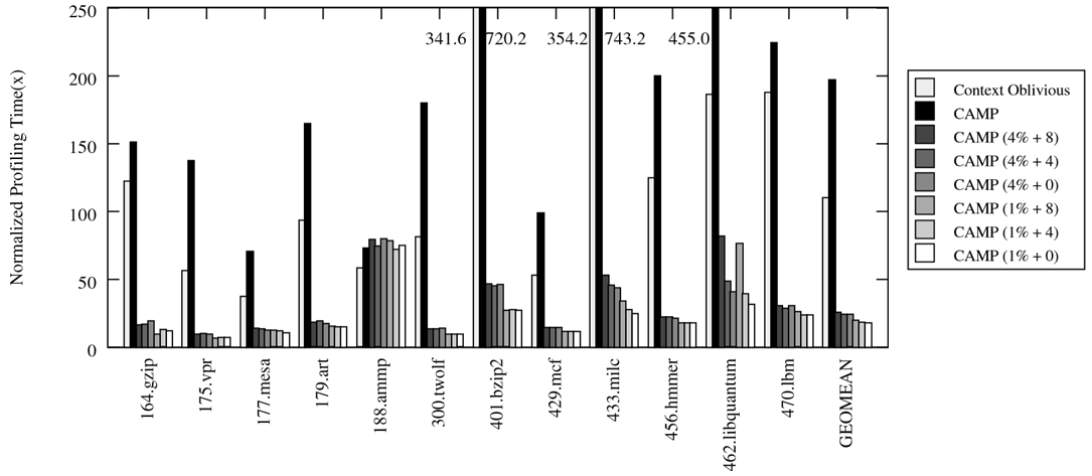
Table 7.1 Benchmark details.[5]

7.1. Time and Memory Overheads of CAMP

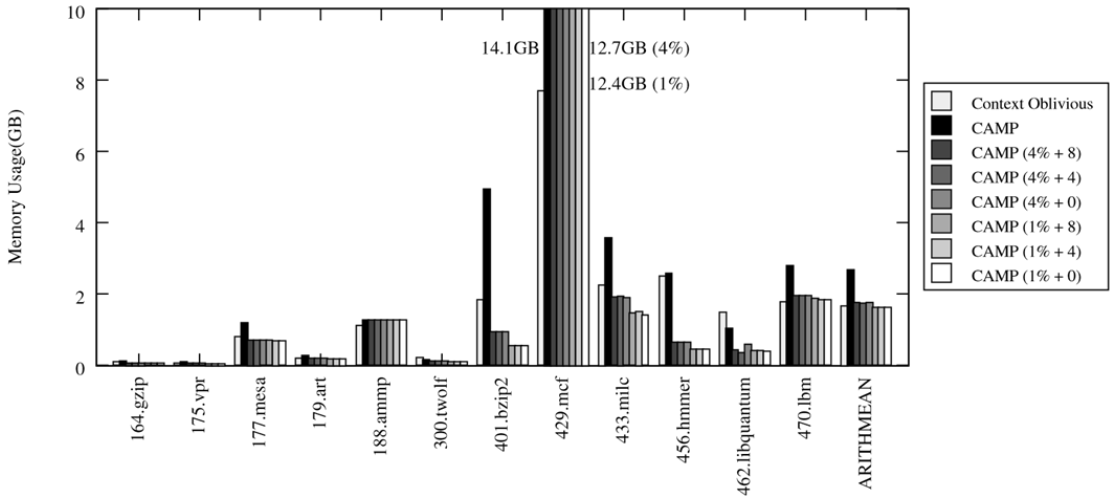
Figure 7.1 shows the whole program profiling time and memory overheads of CAMP. Bases are the execution time and memory usage of the original program without profiling. This paper evaluates a context-oblivious profiler, CAMP without sampling and CAMP with various sampling conditions to analyze the overheads of context-awareness and the effectiveness of the proposed heterogeneous sampling. Here, the context-oblivious profiler is equivalent to CAMP profiler without context-awareness. CAMP with sampling profiles memory instructions at a few initial iterations and randomly

[5] M in the numbers of dynamic instances means millions.

selected iterations of each loop, and all the memory instructions that are not in a loop.



(a) Profiling time normalized to native program execution



(b) Memory usage

Figure 7.1 Profiling time and memory overheads.[6]

[6] Here, context-oblivious, CAMP, and CAMP (4% + 8) mean CAMP profiler

Figure 7.1 shows that CAMP without sampling suffers from 197.0× profiling time and 2.7GB memory overheads on average. Compared to context oblivious profiler, the context-awareness increases profiling time and memory usage by 1.9× and 1.6× respectively. Most of the increased overheads come from generating additional dependencies between the same instructions with different contexts that the context oblivious profiler cannot distinguish, while context management overheads are negligible. Sampling dramatically reduces the profiling time and memory overheads. CAMP that samples memory instructions at initial 4 iterations and 1% randomly selected iterations shows 18.4× profiling time and 1.6GB memory overheads.

For `188.ammp`, there is almost no profiling time and memory overhead difference across context oblivious, CAMP and CAMP with sampling. The `main` function of `188.ammp` invokes a recursive function call, `read_eval_do`, in which most of the program is executed. Since CAMP considers a recursive function call site as a leaf node of a context tree, CAMP creates only 7 context nodes for `188.ammp`, and profiles most of memory instructions with the same context like context oblivious profiling as a consequence. For other programs with recursive function calls such as

---

without context awareness, CAMP profiler without sampling, and CAMP profiler that samples memory instructions at initial 8 iterations and 4% randomly selected iterations of each loop respectively. CAMP (4% + 8) also profiles all the memory instructions not in a loop.

177.mesa, 300.twolf, 429.mcf, 456.hmmer and 462.libquantum, CAMP creates effective context trees and generates precise profiling results.

429.mcf suffers from high memory overheads compared to others. CAMP creates history tables in page size granularity to amortize history table creation overheads with spatial locality. Unfortunately, since 429.mcf sparsely touches memory spaces that span over the page size, CAMP repeatedly and inefficiently creates the history tables instead of reusing existing tables, so CAMP suffers from significant memory overheads for 429.mcf.

7.2. Sampling Accuracy

While sampling memory operations reduces profiling time and memory overheads, sampling compromises precision and sensitivity. To evaluate precision and sensitivity of the proposed heterogeneous sampling, this work measures false positive and false negative of different sampling ratios. Here, precision is the fraction of sampled dependencies that really exist, while sensitivity is the fraction of real dependencies that are sampled. The precision and sensitivity are calculated by equation 7.1. The heterogeneous sampling adopts two different sampling methods such as a consecutive profiling that profiles only a few consecutive initial iterations and a random sampling that profiles randomly selected iterations. CAMP (4% + 8) means that all the memory instructions at initial 8 consecutive iterations and 4% randomly selected iterations of each loop are profiled. The heterogeneous sampling

profiles all the memory instructions not in a loop because the instructions are not repetitive.

$$Precision = \frac{\#\ True\ Positive}{\#\ True\ Positive + \ \#\ False\ Positive}$$

$$Sensitivity = \frac{\#\ True\ Positive}{\#\ True\ Positive + \ \#\ False\ Negative}$$

Equation 7.1. Precision and Sensitivity of Sampling

For all the programs and sampling ratios without any exception, CAMP does not generate any false positive dependency, thus showing 100% precision. Since the heterogeneous sampling updates history tables for all the writes, CAMP correctly finds corresponding memory write history elements for each memory operations. Though the heterogeneous sampling updates history tables for all the writes to guarantee 100% precision, Figure 7.1(a) shows that the sampling still dramatically reduces the profiling time by 10.8×.

The heterogeneous sampling also increases sensitivity by efficiently tracing regular and irregular memory access patterns. Figure 7.2 and Figure 7.1(a) illustrate sensitivity and profiling time of the heterogeneous sampling with different sampling ratios. Profiling a few consecutive initial iterations largely increases the sensitivity by tracing regular memory accesses among consecutive iterations that the random sampling could miss, while the additional consecutive profiling incurs only a small profiling time increase. For example, compared with 1% random sampling only, the additional consecutive sampling for 4 initial consecutive iterations increases the sensitivity by 16.1% at the expense of only 5.0% profiling time increase.

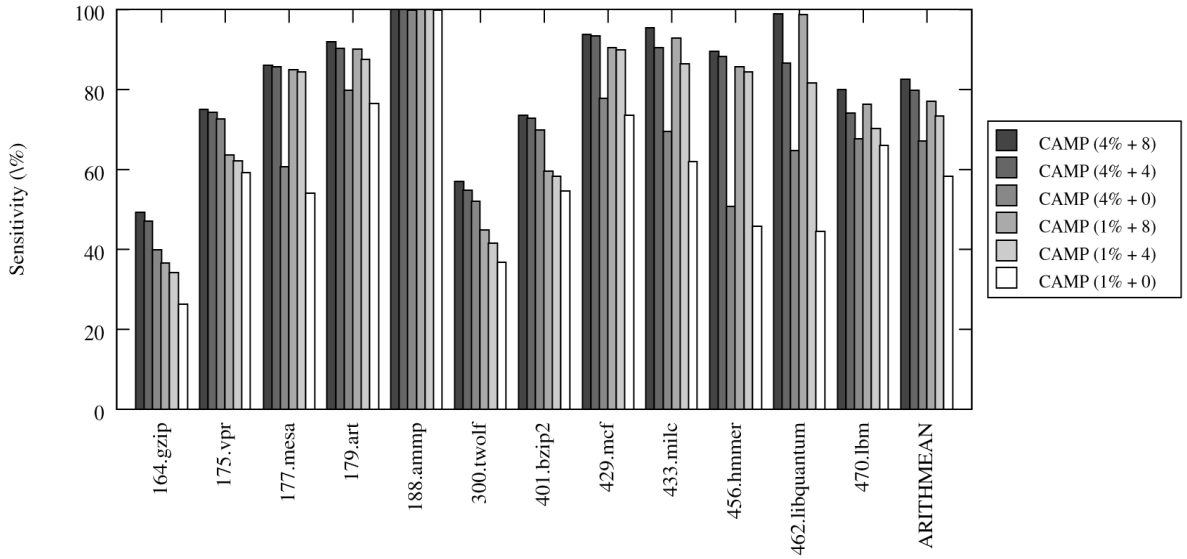


Figure 7.2 Sensitivity of CAMP with different sampling ratios.[7]

[7] Here, precision of CAMP is not illustrated as a graph because the sampling results show 100% precision for all the programs and sampling ratios.

Chapter 8

Related Work

8.1. Context-Aware Memory Profilers

Like CAMP, context-aware memory profilers [4, 18] generate memory dependencies with their contexts such as function call stacks and loop nests. However, none of them fully generates the memory dependencies between all the instructions in a program.

T. Chen et al. [4] 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 dependencies between the current memory instruction and the most recent memory instruction on the same memory, and the compiler reconstructs full memory dependencies 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. [18] 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.

8.2. Loop-Aware Memory Profilers

Loop-aware memory profilers [8, 10, 11, 22, 24] 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 [11] 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. [10] proposed SD3 profiler that is a parallel memory profiler. SD3 reduces profiling time overhead with parallel profiling, and also reduces memory usage overhead with data compression using frequent loop-stride characteristics of computational program. Since each memory dependence generation in CAMP is independent of 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. [24] 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. [22] 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 tool profiles only pre-selected instruction sets.

A. Ketterlin et al. [8] 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.

8.3. Context Management in Profilers

Context management of CAMP is highly inspired by previous context-aware performance profilers [2, 6, 27]. G. Ammons et al. [2] first introduced a call tree in which each node reflects a call site. Adaptive calling context tree profilers [6, 27] support sampling-based calling context management to reduce performance overhead. Unlike the previous profilers [2, 6, 27], CAMP constructs a context tree for every function call site and loop invocation. 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.

Chapter 9

Conclusion

In order to make precise context-aware PDGs, this paper proposes a context-aware memory profiler (CAMP) which traces memory dependencies with their full context information. As a compiler-runtime cooperative system, CAMP utilizes a static context tree to make concise representations for every obtainable context in a program. At profiling time, these concise representations enable efficient discovery of context-aware dependencies. Regarding the resultant PDGs, CAMP discovers that 70.8% of total dependencies that a context oblivious profiler makes are false; it allows us to deny a significant number of false dependencies which stem from ignorance of contexts, thus resulting in more precise PDGs. Through a case study, we show that how a precise context-aware PDG facilitates a compiler optimization such as speculative parallelism. With the heterogeneous sampling method, CAMP finds 73.3% of all possible memory dependencies at the finest granularity (i.e. instruction-pairwise and byte-level), while suffering from only 18.4× slowdown for 12 programs from SPEC, which is considered acceptable in practice.

요약문

프로그램에 존재하는 데이터와 컨트롤의 의존관계를 표현하는 Program Dependence Graph(PDG)는 프로그램 분석에 있어서 핵심적인 역할을 해왔다. 특히, 프로파일링(profiling)에 의해 동적으로 생성된 Speculative PDG 는 자동 병렬화(automatic parallelization)와 같은 공격적인 최적화 기법에 필수적으로 쓰인다. 예컨대, [9, 13]에서는 루프에 존재하는 동적 의존 관계를 메모리 프로파일러로 측정하여, 런타임에 자주 발생하지 않는 의존 관계를 무시함으로써 프로그램을 병렬화 하였다.

그런데 PDG 를 프로파일러가 동적으로 생성할 때에는, 함수 호출 위치(function call site)와 루프와 같은 콘텍스트(context) 정보를 담고 있어야 더욱 정확한 PDG 를 만들 수 있다. 만일 콘텍스트 정보가 없으면 다른 콘텍스트에서 실행된 같은 명령어 쌍을 구분할 수 없게 되어 수 많은 거짓 의존관계를 낳게 된다. 예컨대, 루프에 대한 콘텍스트 정보가 없다면, 루프 반복 구간 안에서 발생하는 의존 관계(intra-iteration dependency) 와 반복 구간 사이에

발생하는 의존 관계(inter-iteration dependency)를 구분할 수 없게 된다. 이와 같은 문제를 해결하기 위해서는 콘텍스트를 인지하는(context-aware) 프로파일러를 만들어야 한다.

본 연구에서는 더욱 정확한 PDG를 생성하기 위해, 콘텍스트를 인지하는 프로파일러인 CAMP를 제안한다. CAMP는 프로그램에서 발생하는 모든 메모리 의존 관계를 byte 단위로 측정하며, 각 의존 관계마다 모든 콘텍스트 정보를 빠짐없이 기록한다. CAMP 컴파일러는 프로그램의 구조를 정적으로 분석하여, 프로그램에서 존재할 수 있는 모든 콘텍스트를 표현하는 콘텍스트 트리(Context Tree)를 만드는데, 이는 CAMP의 런타임 시스템이 단 하나의 동적 콘텍스트 ID만으로 다양한 콘텍스트를 계산하는 것을 가능케 한다. CAMP 런타임은 프로그램의 실행 콘텍스트가 변화할 때마다 동적 콘텍스트 ID에 정적 콘텍스트 ID를 더하는 간단한 산술연산을 한번만 취함으로써 바뀐 콘텍스트를 계산한다. 새로운 콘텍스트에서 읽기(load), 쓰기(store)와 같은 메모리 접근이 관측되면 해당 메모리 주소에 접근한 과거 기록을 사용하여 메모리 의존 관계를 계산하며 그 결과를 콘텍스트와 함께 저장한다. 결과적으로, 콘텍스트 트리는 CAMP의 프로파일링 시간을 단축시키고 메모리 오버헤드를 완화하는데 핵심적인 역할을 한다.

프로파일링 오버헤드를 추가적으로 줄이기 위해서 CAMP 는 두 가지 샘플링 기법을 결합하여 사용한다. 첫 번째 샘플링 기법은 루프의 반복 구간을 무작위로 선정하여 선정된 구간에서만 메모리 접근을 관측하는 무작위 샘플링(random sampling)이다. 두 번째 샘플링 기법은 루프의 처음 반복 구간 몇 차례만 관측하여 루프에 존재하는 줄무늬(stripe) 패턴의 의존 관계를 찾아내는 연속적 샘플링(consecutive sampling)이다. 두 샘플링 기법을 결합하여 사용함으로써 루프에 존재하는 대부분의 의존관계를 찾아 내었다. 또한, 거짓 양성(false positive)이 발생하는 것을 방지하기 위해, 샘플링 여부와 상관없이 쓰기(store) 연산이 발생하면 메모리 접근 기록 테이블 (History Table)을 초기화 하였다.

SPEC 벤치마크[1]에 대하여 실험한 결과, CAMP 를 사용하면 의존 관계를 더욱 정확히 찾아낼 수 있을 뿐만 아니라 더 많은 루프를 병렬화 할 수 있다. CAMP 는 콘텍스트를 인지하지 못하는 (context-oblivious) 프로파일러가 만들어내는 모든 의존 관계 중 약 70.8%가 거짓(false positive)이라는 것을 밝혀냈다. CAMP 를 사용하면 PDG 를 더욱 간결하고 정확하게 나타낼 수 있고, 이렇게 생성된 PDG 를 사용하면 병렬화가 가능한 루프를 9.7%만큼 추가로 찾아 낼 수 있다. 401.bzip2 의 경우에는 병렬화 할 수 있는 루프가 54.2%만큼 증가한다.

이 연구의 주요 성과를 요약하면 다음과 같다.

- 프로그램에 존재하는 의존관계를 콘텍스트 정보 손실 없이 관측할 수 있도록 프로파일러를 만듦
- 컴파일러의 정적 분석(static analysis)을 통해 프로그램에 존재할 수 있는 모든 콘텍스트를 간략하게 표현할 수 있는 콘텍스트 트리 개발하였고 이것으로 프로파일링 오버헤드를 완화시킴.
- 서로 다른 샘플링 기법을 결합하여 거짓 양성 관측 없이 대부분의 의존관계를 찾아낼 수 있는 새로운 샘플링 기법 제안
- SPEC INT2000 과 SPEC2006 을 사용한 심층 분석을 토대로 콘텍스트 정보를 포함한 speculative PDG 의 장점을 분석

REFERENCES

[1] Standard Performance Evaluation Corporation. http://www.spec.org.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] M. Kim, H. Kim, and C.-K. Luk. SD3: A scalable approach to dynamic data-dependence profiling. In Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, 2010.

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

[12] 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.

[13] 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.

[14] 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.

[15] C. G. Qui˜nones, C. Madriles, J. S´anchez, P. Marcuello, A. Gonz´alez, 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.

[16] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative runtime parallelization of loops with privatization and reduction parallelization. IEEE Transactions on Parallel Distributed Systems, 1999.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] P. Wu, A. Kejariwal, and C. Cas¸caval. Languages and compilers for parallel computing. chapter Compiler-Driven Dependence Profiling to Guide Program Parallelization. 2008.

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

[25] 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.

[26] 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.

[27] 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.

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

Acknowledgements

감사의 글

포항공과대학교에 입학한 것이 엊그제 같은데 어느덧 졸업을 하게 되었습니다. 우선 학부시절부터 저를 지도해 주시고 대학원 과정까지 이끌어주신 저의 지도교수님이시자 멘토이신 김한준 교수님께 감사 드립니다. 교수님 덕분에 학문뿐만 아니라 제가 살아가는 방식까지 다시 돌아보고 성장할 수 있게 되었습니다. 또한, 바쁘신 와중에도 석사 학위 논문 심사를 흔쾌히 수락해 주시고 아낌없는 조언을 해주신 김장우 교수님과 배경민 교수님께 감사 드립니다.

컴파일러 연구실의 출발부터 연구실이 성장해 나가는 과정을 함께 할 수 있었던 것은 정말 좋은 경험이었습니다. 정신적으로, 학문적으로 힘들 때 항상 많은 도움이 되어 주었던 컴파일러 연구실 동료들에게 감사의 말을 전하고 싶습니다. 연구실에서 동고동락하며 서로에게 큰 힘이 되어준 봉준이형, 선영누나, 창수, 주원이에게 모두 감사합니다. 친구 같은 후배인 준하, 승빈이에게 고마움을 전하며 앞으로의 연구가 좋은 성과를 맺기를 기원합니다. 학부 연구 참여 때부터 많은 도움을 주고 함께 힘든 과정을 해쳐나갔던 경민이, 또 먼저 졸업하신 현준이형, 경주누나에게도 감사함을 전하고 싶습니다. 연구 주제가 다르지만 학문적으로 많은 조언을 준 광무형, 다열이형, 동업이형에게 고맙습니다. 그리고 룸메이트이자 인생 선배로서 정신적인 지주가 되어주셨던 동훈이형에게 진심으로 감사하다는 말을 전하고 싶습니다.

마지막으로 저를 언제나 신뢰해주시고, 사랑해주시고, 공부에 집중할 수 있도록 항상 물심양면으로 키워주신 부모님께 감사드립니다. 졸업 후에도 더욱 성장하고 발전하여 새로운 가치를 창출해 낼 수 있는 멋진 사람이 될 수 있도록 노력하겠습니다. 감사합니다.