Data Dependence Profiling for Speculative Optimizations

Tong Chen, Jin Lin, Xiaoru Dai,Wei-Chung Hsu, and Pen-Chung Yew

Department of Computer Science, University of Minnesota
{tchen, jlin, dai, hsu, yew}@cs.umn.edu

Abstract. Data dependence analysis is the foundation to many reordering related compiler optimizations and loop parallelization. Traditional data dependence analysis algorithms are developed primarily for Fortran-like subscripted array variables. They are not very effective for pointer-based references in C or C++. With more advanced hardware support for speculative execution, such as the advanced load instructions in Intel's IA64 architecture, some data dependences with low probability can be speculatively ignored. However, such speculative optimizations must be carefully applied to avoid excessive cost associated with potential mis-speculations. Data dependence profiling is one way to provide probabilistic information on data dependences to guide such speculative optimizations and speculative thread generation. Software-based data dependence profiling requires detailed tracing of memory accesses, therefore, could be very time consuming. In this paper, we examine issues related to data dependence profiling, and propose various techniques to improve the efficiency of data dependence profiling. We use the Open Research Compiler (ORC) [15,16] to test the efficiency of our data profiling techniques. We also study the effectiveness of data dependence profiling on data speculative optimizations on Itanium systems. Our results show that efficient data dependence profiling could improve the performance for data speculative optimizations.

1 Introduction

General data dependence analyses [1, 2, 3] have been extensively used for Fortran-like subscripted array expressions in scientific applications. However, those algorithms are mostly inadequate for C and C++ programs due to the widely use of pointer expressions. Existing approaches usually require a very sophisticated inter-procedural pointer alias analysis followed by a shape analysis [4, 5, 6]. The effectiveness of such approaches is often limited because of the increasing use of shared libraries, complex recursive data structures, and dynamically allocated objects in applications [7, 8].

To circumvent such limitations in compiler analyses, some architectural and hardware schemes have been proposed to support data dependence speculation. For example, Intel's IA64 architecture [9] provides the *advance load* instruction which can be used to schedule a *load* instruction across a possibly aliased *store* instruction. The correctness of the execution is enforced by checking whether such dependences exist or not at runtime. If no such dependence exists, the speculation is successful. If such dependence does exist, the check fails and a respective recovery action will take place. A similar check and recovery approach has been proposed in speculative multithreaded processors [10, 11].

Like other speculative execution, the cost of mis-speculation is relatively high. To make use of the data speculation, the compiler typically assumes that the mis-speculations should be very rare. However, it is often very difficult, if not possible, for a compiler to obtain such information based only on static analyses algorithms. Simply ignoring all possible data dependences may cause drastic performance degradation in some programs due to frequent mis-speculation and recovery. Like many existing profile-guided optimizations, data dependence profiling could provide the compiler with useful guidance on data speculative optimizations.

In this paper, we present a software-only data dependence profiling approach to study the potential of data dependence profiling. Our data dependence profiling tool is instrumentation based. It is platform independent and can be used for many different types of data speculative execution. Although hardware supported data dependence profiling [12, 13] can be more efficient, they usually are limited by the size of the hardware table, and can only track data dependence among references in a constrained window of execution. Our tool does not have such limitations. It could provide data dependence information for a larger code region, not limited by program structures such as basic blocks, function calls, or loops. The instrumentation-based profiling approach could also provide additional flexibilities. The probability of a data dependence edge, for example, can be defined and, consequently, collected for different optimizations.

Alias profile has been used to guide speculative register promotion in [14]. However, the information provided by the alias profiling is usually less accurate than that by the data dependence profiling. For example, two pointers p and q both point to a memory block allocated from the same *malloc*, but access different parts of the memory block. The alias profiling may indicate that p and q are all pointing to the same memory objects. However, data dependence profiling is based on pair-wise *address* comparison, so that *p* and *q* will not be data dependent on each other. We have conducted experiments to show that using data dependence profile yields a higher performance gain than using alias profile on the same speculative optimizations. Furthermore, data dependence profile can be more useful in parallelization than the alias profile. This is because disambiguation among array elements requires finer granularity than a typical alias profile can provide.

Some unique features of our data dependence profiling tool are listed below:

- It detects data dependence quickly. To detect data dependence at runtime, the addresses accessed by each pair of memory references need to be compared. Such cross checking has a complexity of $O(n^2)$, where n is the number of references. Other profiling approaches, such as edge profiling or value profiling, usually have a complexity that is linear to the profiling events. Hence, *efficiency* is very important in data dependence profiling, especially for very large application programs. We use a special hash function in conjunction with some sampling techniques to ensure data dependences can be detected very quickly.
- It handles function calls. Our profiling tool detects data dependences among references within the same function calls, as well as data dependences across function calls. The side effect of function calls can be summarized, and the

compiler can use such information to overcome the barrier of function calls during its optimizations.

- It handles nested loops. For loop related optimizations, it is very important to distinguish loop-carried dependences from loop-independent dependences. In our data dependence profiling, we associate each dependence edge with a distance vector. Since we do not want to limit the collection of data dependence information to the innermost loop only, the profiling tool can generate distance vectors for nested loops simultaneously. Notice that the loops may be nested across procedure boundaries.
- It tracks the dynamic behavior of data dependence in programs. A data dependence edge may occur with different *probabilities* at runtime. However, we found that the *probability* of a data dependence edge needs to be well defined according to the consumers of the profiling information. How to collect the *probability* information in profiling needs to be addressed.

We have implemented our data dependence profiling tool in Intel's Open Research Compiler (ORC) [15, 16]. The collected data dependence profile can be fed back to the ORC compiler for speculative optimizations. The main contributions of this paper are listed below:

- We present the design and the implementation of an instrumentation-based data dependence profiling tool. This tool is capable of generating detailed data dependence profiles (such as *dependence distance vectors* and *dependence probability*) for nested loops. Different dependence probabilities can be defined and profiled accordingly for different speculative optimizations.
- We discuss and evaluate several techniques to improve the efficiency of data dependence profiling. We use a *shadow* memory space to conduct efficient data dependency checking. Since profiling is merely an approximation of the program behavior, we evaluate various implementation tradeoffs and their impact on the quality and the efficiency of the data dependence profiling. Various sampling techniques to reduce profiling overhead is also studied. Since data dependence is defined among memory references, it needs to consider the program structures, such as procedures and loops. Furthermore, an implementation of sampling technique similar to [17] has reduced the profiling overhead from many times of the original program execution time down to only 20%.
- We show the benefit of data dependence profiling in two optimizations: profile-guided code scheduling and speculative partial redundancy elimination. The performance improvement of some SPEC CPU2000 benchmarks on Itanium machines can be as high as 32%.

The rest of this paper is organized as follows: the benefit of data dependence profiling is shown in section 2; Section 3 describes how the data dependence profiling is performed; The experimental results on how to reduce the overhead of data dependence profiling are presented in section 4; Section 5 discusses the related works; The final section provides the conclusions of this paper.

2 Benefit from Data Dependence Profiling

Before we describe how data dependence profiling is performed, we would like to first show its potential benefit to two common compiler optimizations: the partial redundancy elimination and code scheduling. Our instrumentation tool is built on top of Intel's Open Research Compiler (ORC). The instrumented code is then linked with our data dependence profiling library to generate an executable file. Data dependence is profiled when this executable file is executed with *train* input set. No space reduction or sampling techniques (discussed later) are applied. The profiling results are then fed back to ORC to guide optimizations. The optimized code is executed with the *ref* input set and the execution time is measured. Our baseline is the code compiled with ORC -O3. The reduction in the execution time is reported as performance improvement. The machine used in our experiments is HP workstation i2000 with one 900MHz Itanium2 processor and 2GB of main memory. Our test programs are SPEC CPU2000 benchmarks [18].

2.1 Speculative PRE

The partial redundancy elimination (PRE) [19] includes a set of compiler optimizations such as register promotion, strength reduction and expression redundancy elimination. It is one of the most important optimizations in the compiler. Data speculation has been introduced into PRE using alias profiling information [20, 21]. However, alias profiling often cannot disambiguate among array references, or among memory references in a memory space managed by the application program itself. It also cannot determine whether aliases generated come from the same iteration or from different iterations. Instead, data dependence profiling often can provide more detailed information about memory references than alias profiling.

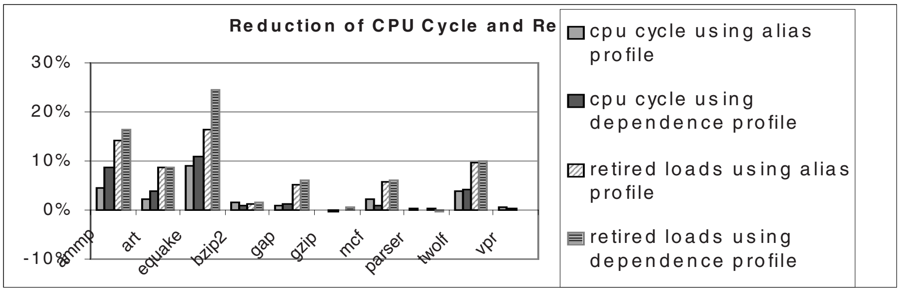

Fig. 1. Improvement on PRE using dependence profiling vs. alias profiling

The speculative PRE using both alias and data dependence profiling information has been implemented on ORC. We measure its performance using *pfmon* [22] on Itanium systems. The performance improvement based on alias profiling and dependence profiling is compared in Fig. 1. The total number of CPU cycles and the total number of *loads* retired are measured for each program optimized by the speculative PRE. The result shows that data dependence profiling is able to discover more redundant loads (more retired loads reduced) and outperforms alias profiling in the total CPU cycle.

2.2 Speculative Code Scheduling

Code scheduling aims to exploit ILP. Data speculation has been used to move *loads* speculatively so that the length of a critical path can be reduced [23]. In the ORC compiler, the speculation is based on heuristic rules. For example, a *load* instruction can be moved across only two *aliased store* instructions (based on the alias analysis in the compiler). We try to conduct more aggressive data speculation based on the data dependence profiling information. Dependence edges with a probability lower than 2% are ignored. This threshold is determined by the overhead of a data mis-speculation on Itanium2: it takes about 50 cycles to jump to and return from its recovery code at a check failure.

Table 1. Improvement on code scheduling using dependence profiling

	Performance improvement	Regular loads changed to speculative loads	Failure rate
equake	8.36%	18.80%	0.34%
art	32.34%	31.32%	0.88%
Mesa	12.23%	9.14%	0.09%
Bzip	0.34%	6.81%	6.98%
Gzip	0.00%	2.45%	0.11%
parser	2.36%	2.92%	2.00%
Vortex	1.94%	4.62%	0.13%
Average	8.22%	10.8%	1.5%

The first column of Table 1 shows the performance improvement on Itanium2 using data dependence profiling information. Some floating-point benchmarks such as *art* can improve performance by 32%. On average, 11% of regular *load* operations are turned into *speculative loads* (reported in the second column) with a very low check failure rate (i.e. less than 1% of mis-speculation) for most benchmark programs, except *bzip* and *parser*. The major reason for the improvement is that, ORC, in fear of the high penalty of mis-speculation, is overly conservative on scheduling data speculative loads. With dependence profiling, more aggressive scheduling can be performed with confidence. The failure rate observed here also shows that the dependence profiling is quite insensitive to different input sets. When we ignore the dependence with less than 2% probability observed with *train* input, only bzip2 has a failure rate higher than 2% for the execution with *ref* input.

3 Instrumentation-Based Data Dependence Profiling

In this section, we address the issues related to data dependence profiling that include detection of data dependences, handling of function calls and nested loops, and the probability of a data dependence edge.

3.1 Detection of Data Dependences Using Shadow Variables

In our data dependence profiling, we focus only on the data dependence among *memory references*. We ignore data dependence among *registers* because it can be easily obtained by simple static analysis. Therefore, only memory references are instrumented. The instrumented profiling instructions collect the *address value* and *the reference ID* of each instance of memory references at runtime. The reference ID is used to identify each static reference. It is assigned during the instrumentation so that profiling results could be mapped back to the compiler.

Data dependence occurs when two memory references access the *same memory location* in the data space. One occurrence of data dependence can be represented as a *dependence edge* from the *source* to the *sink.* There are four data dependence types: *flow dependence* (or true dependence*), anti-dependence*, *output dependence*, and *input dependence* (or reference dependence).

It is very expensive to check data dependence based on pair-wise address comparison, especially when the program has a large number of memory references. The number of dynamic instances of all memory references could be very large when there are nested loops in the program. Therefore, we use a special data structure, called *shadow* memory, to efficiently detect data dependences. This is a typical trade off between *time* and *space*.

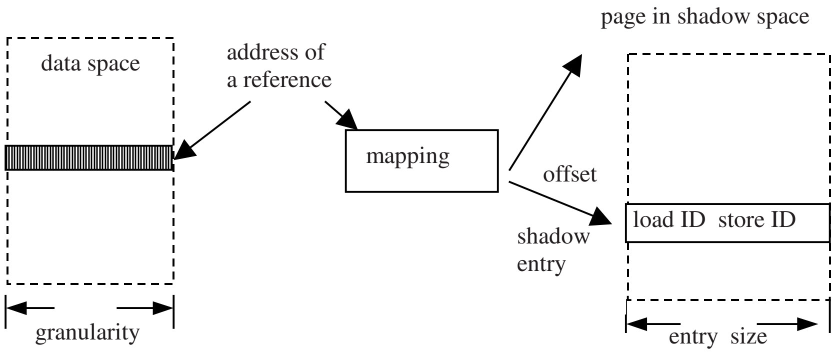

Fig. 2. Detect data dependence using shadow

Shadow is used to store information during profiling. A simple hash function using the *address value* maps each memory reference in the data space to its corresponding shadow entry in the shadow memory (illustrated in Fig. 2). The shadow memory is allocated on demand, and can be freed when the profiling for a particular region such as a particular nested loop or a procedure of interests has completed.

Now, we describe how to detect data dependence during the profiling process. First, each memory reference locates its shadow entry using the hash function on the address value. The shadow entry contains the reference ID of the *latest load* and the *latest store* operation to this memory location, or is *empty* when this location is accessed for the first time. If the memory reference is a *load* operation, there should be a *flow* dependence edge from the latest store to this load operation, and also an *input* dependence edge from the latest load to this load operation. If the memory reference is a *store* operation, there should be an *anti* dependence edge from the latest load to this store operation, and also an *output* dependence edge from the latest store to this store operation. Finally, the reference ID of the current memory reference is stored into the shadow entry, and the reference ID for previous load or store in the shadow entry is overwritten.

Using this scheme, data dependences can be quickly detected without expensive pair-wise address comparisons among all memory references. Using shadow space is equivalent to a software implementation of the associative memory in [21]. We reduce the number of pair-wise comparisons by focusing only on the execution path of a program, i.e. only the latest load and the latest store are compared. We can use a linked list, for each type of data dependence edges, to record the dependence edges.

3.2 Function Calls

When function calls are present in a profiled program, a dependence edge detected between two memory references in two different procedures needs to be mapped into the common procedure that contains both procedures. It is because, to be useful to a compiler, a data dependence graph of a procedure should contain only the dependences among the memory references and function calls, within the same invocation of the procedure. However, profiling information at runtime may contain a lot of dependences between different invocations of procedures at different call sites. We need to sort through these data dependences and extract relevant data dependence information. It requires the calling path information of all memory references to avoid creating *false* dependence edges.

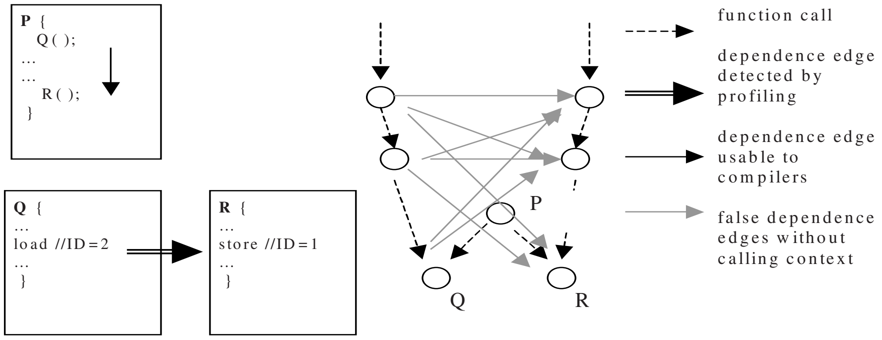
P {
Q( );
...
...
R( );
}
Q {
...
load //ID=2
...
}
R {
...
store //ID=1
...
}
P
Q
R
function call
dependence edge detected by profiling
dependence edge usable to compilers
false dependence edges without calling context

Fig. 3. False dependences may be introduced by function calls

In Fig. 3, for example, a flow dependence from reference 2 to reference 1 occurs only when procedure Q and R are called within procedure P. Therefore, only one dependence edge from the corresponding call site of Q to the call site of R in the procedure P should be generated. If we do not have calling path information for each memory reference, we have to assume that there is a dependence edge from any call site that may calls Q to any call site that may calls R. Here, we include all the procedures calling Q or R directly or indirectly. They are shown as *false* dependence edges in Fig. 3. The example illustrates the need for calling path information in data dependence profiling.

To reduce the number of false dependence edges caused by function calls, the calling context for each memory reference is recorded in its shadow entry. When a dependence edge is detected, their calling contexts help to locate the common procedure. Different kinds of calling context can be used in the dependence profiling, resulting in different precision and efficiency.

- Fully extended call paths. Whenever a procedure is called, a new call node is assigned for this invocation. The call path made up with such call nodes is able to distinguish both the different call sites of a procedure and the different invocations of a call site. It is easy to maintain the fully extended calling path at runtime: push a new node to the call stack when a procedure is invoked, and pop the top node from the call stack when a procedure call returns. The nodes in the call stack can be linked backwardly and we only keep the top node in the shadow entry. When a dependence edge occurs, the common part in their calling context can be identified. A dependence edge should be added in the farthest common procedure from the main procedure between the corresponding references or call sites.
- Compacted call paths. The space requirement of the fully expanded call paths could be very high because a call site may appear in a nested loop or in a recursive call. We could reduce the size of a call path by using only one node to represent multiple invocations of a call site in a nested loop or in recursive calls to reduce the space requirement. Compression techniques [24] can also be used. However, compacted call paths may result in some false dependence edges.

For SPEC CPU2000 benchmarks, even with the train input set, it is still too costly to fully extend all call paths. We use the compact call paths for efficiency.

3.3 Loops

When a dependence edge occurs in a loop nest, a *dependence distance* tells how far the sink is away from the source of a dependence edge in terms of the number of iterations. Optimizations may treat dependences with different distances quite differently. It is important for a profiling tool to be able to generate distance vectors.

In order to generate a distance vector for a dependence edge, we instrument the beginning of the loop body. Each loop is also assigned an iteration counter. This counter is incremented when the beginning of the loop body is encountered. The values in the iteration counters of the loops nested outside of the current reference form an iteration vector. We store the iteration vector in the shadow. When a dependence edge is detected, we first have to find out how the two references nested in loops, with the consideration of the calling context discussed in the previous section. The loop nesting information is not recorded in the shadow. The calling context information in the shadow is used to help identifying the commonly nested loops of the two references. The iteration vectors of the two references are aligned and the iteration counters of their commonly nested loops are identified. The distance vector is computed by subtracting the iteration vector in the shadow from the current iteration vector. Dependences together with their distance vectors are recorded. Depending on the optimizations using the profile, limited distance or the sign of the distance, =, < or >, can be recorded instead for better efficiently.

3.4 Dependence Probability

A static dependence analysis in a compiler usually can only tell whether an edge between two memory references exists or not. However, with the dependence profiling, we can further observe whether the data dependence *rarely*, *frequently*, or *always* occur between the two memory references. Such additional information allows a compiler optimization to deal with a data dependence edge accordingly for a better performance. In general, a compiler can speculate only on those dependences that *rarely* occur in order to avoid costly mis-speculations. Hence, such information is especially important to speculative optimizations in a compiler.

We use a *dependence probability* to depict such dynamic behavior of the dependence. There are several ways to define the *dependence probability*. It depends on how such information is to be used by later speculative optimizations. Here, we only focus on two possible definitions: the *reference-based probability* and the *iteration-based probability*.

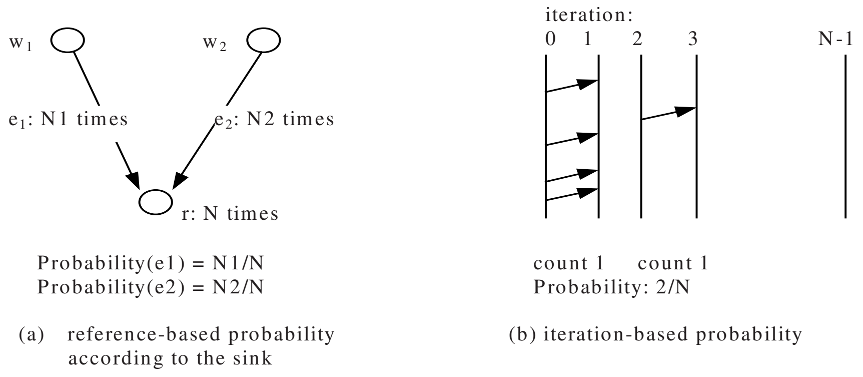

Fig. 4. Different definitions of dependence probability for dependence profiling

The *reference-based probability* is defined as the number of the occurrences of a *dependence edge* over the total number of the occurrence of the *reference*. The *reference* may be either the *source* or the *sink* of the *dependence edge*. This information gives an indication of how often a dependence edge occurs when either the source or the sink of the dependence is referenced during the program execution. In Figure 4(a), an example of the reference-based probability for the sink reference is illustrated. Assume e_1 and e_2 are two flow dependence edges to the sink reference r. The reference r is executed N times while the edges e_1 and e_2 occur N_1 and N_2 times respectively. The probability of the value coming from reference w_1 is defined as N_1/N and from reference w_2 N_2/N. Such probability is able to tell us which definition is more likely to provide the value to the reference r.

The *iteration-based probability* is defined as the *number of iterations* in which the dependence edge occurs over the total number of iterations of the loop nest, as shown in Figure 4(b). Again, the *iteration-based probability* may be *sink-based* or *source-base*. This information is often related to the dependence distance and the control flow within the loop.

4 How to Reduce the Profiling Overhead

From the discussion in section 3, the data dependence profiling could be quite time consuming because we need to handle each memory reference individually, and there is a significant amount of work on each memory reference. In this section, we present ways to reduce such overhead during the dependence profiling.

The most direct way to reduce such overhead is to reduce the number of memory references that need to be instrumented and tracked. In a typical optimizing compiler such as ORC, many of its optimizations already work toward reducing the number of memory references in the application program. Therefore, our instrumentation is able to make use of the existing optimizations.

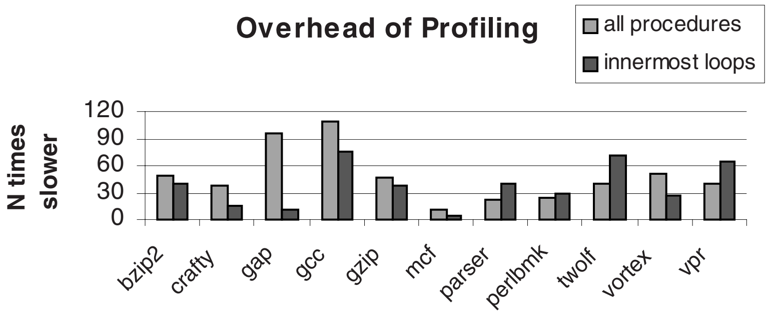

Fig. 5. Overhead of dependence profiling

The data dependence profiling is still quite expensive even when the instrumentation is done after WOPT, in which ORC eliminates most of the redundant memory references. The overhead of various dependence profiling is shown Fig. 5. The average overhead is over 40 times of the original execution time even only for the innermost loops. We need to somehow improve the space and time efficiency to make the tool more usable.

Unfortunately, the tradeoff to reduce the overhead of profiling often involves in losing some *precision* in data dependence profiling. In the following experiments, we will measure both the *efficiency* and the *precision* of each scheme proposed. The efficiency can be measured by the *space requirement* and/or the *profiling time*. We also need to address the precision issues in more details. First, we can measure how many false dependence edges are produced and/or how many true dependence edges are missing after a more efficient profiling method is used. The percentage of such dependence edges with respect to the total dependence edges can be used for such a measure. Secondly, the *dependence probability* of such false/missing dependence edges should also be considered. A dependence edge with lower probability will have a less impact on optimizations because the compiler may speculate it as non-existence.

4.1 Reduce Shadow Size

As described in section 3.1, we trade space for time using shadow during dependence detection. It is important to reduce the shadow size for better space efficiency. We define the hash function for the shadow in such a way that the conflict in hashing rarely occurs. The total shadow size is determined by: *(size of program space in byte)*(size of a shadow entry)/granularity*. The *granularity* is determined by how many adjacent data bytes would share *one* shadow entry, while the *shadow entry size* is determined by how many bytes are need to keep the runtime information during the profiling (see Fig. 2). While the program space cannot be changed, the profiling tool can manipulate the *granularity* and the *shadow entry size*.

Since most of the memory references access 4 bytes or 8 bytes each time, the size of total shadow space can be reduced if we let 2^k adjacent data bytes share a shadow entry. However, larger granularities may introduce more false dependences.

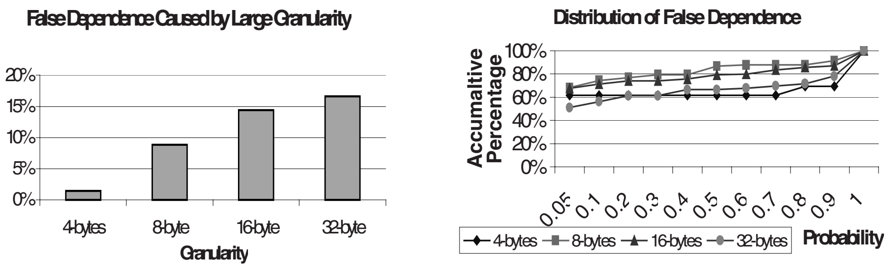

Fig. 6. Precision of different granularities

To study the impact of the granularity, we compare the profiling results with a granularity of 4, 8, 32 and 64 bytes to the results with a granularity of 1 byte in Fig. 6. The results show that the granularity of 4 bytes is quite satisfactory. The reason is that 4-byte data, such as an integer or a 32-bit floating-point number, are the most commonly used. The precision of using a granularity of 16 or 32 bytes is still acceptable. This means that using a cache line for profiling or for communication is still a good compromise.

Another factor affecting the size of shadow is the size of each entry in shadow. From the previous discussion, we know that the reference ID, the calling path and the iteration vector of each reference are stored in each shadow entry. Since compression techniques have been applied to the reference ID and calling path, we focus on the iteration vector here. To uniquely represent iterations of the loops in SPEC2000 benchmarks, the iteration counter in an iteration vector has to be 8 bytes. Hence, a deeply nested loop may need a huge shadow entry. Since only dependences with a short distance, usually a distance of 0 and 1, are important to optimizations, we can store only a partial value of an iteration counter in the shadow. As a result, some false dependence edges may be generated due to the overflow of iteration counters. Fig. 7 reports the precision of iteration counters with a size of 1, 2, and 4 bytes. In this experiment, only the important loops (i.e. the most time-consuming loops) are selected, and only the flow dependence edges with a distance of 0 or 1 are profiled. Though the 1-byte iteration counter may result in a very high percentage of false dependences, most of these false dependences have a probability close to 0, and, hence, could be speculated with little impact on performance. This result helps us to choose 1 byte as the size of our iteration counters.

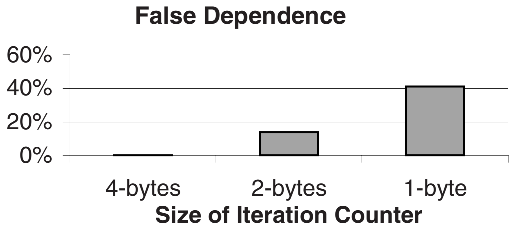

Fig. 7. Precision of different size of iteration vector

4.2 Use Sampling to Improve Time Efficiency

Time efficiency of data dependence profiling could be improved by sampling techniques. In sampling, only a small portion of the events is selected to reduce profiling overhead. The sampling technique has been widely used in many other profiling applications, such as edge profiling, execution time profiling and alias profiling. In these profiling, sampling is directly applied to the relevant events. For example, in edge profiling [25], the direction of each branch (i.e. branch *taken* or *not taken*) is sampled. However, such sampling is not suitable for data dependence profiling. One reason is that the data dependence is a *relation* between two memory references, i.e. its *source* and *sink*. We cannot sample them *independently*. Second reason is that the number of occurrences of a data dependence edge is not directly related to its significance to a certain compiler optimization. Randomly sampling *individua*l references may not work for data dependence profiling from a statistical point of view. More organized sampling such as considering a program segment as a whole for sampling is more appropriate. A program segment can be a procedure or a loop.

Most important procedures or loops in our benchmarks are executed many times because they are usually contained in other loop structures and are the most time consuming parts of the program. We can thus use the framework proposed in [17] for more efficient sampling. In this framework, we keep two versions of the program segment to be sampled: one is *with* and the other is *without* instrumentation for dependence profiling. A switch point is set at the entry and the exit of the program segment. We will then switch between these two versions during the execution of the program with a pre-determined sampling rate. Since the profiling is done on the *whole* program segment, instead of *individua*l memory references, a more complete snap shot of the data dependences in the program segment can be obtained.

We tested such a sampling technique for both procedures and loops. Different sampling rates are used. The profiling overhead with different sampling rates is measured against the original execution time. The precision of the profiling is measured using the number of missing dependence edges. Such missing dependence edges may in turn introduce some false dependence edges due to the transitive nature of the data dependences. These false edges will not affect the program optimizations. Fig. 8 reports the overhead and the precision of different sample rates for procedures. They are usually in the range from 16% to 167% of the original execution time using a sampling rate of 0.0001 to 0.1, respectively, and with a precision ranging from 30% to 10% in missing dependence edges.

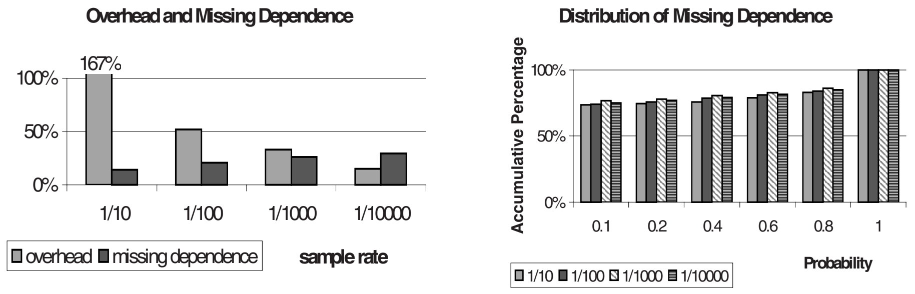

Fig. 8. Sampling of the dependence in procedures

A loop in a procedure may also be invoked many times, and each invocation may have only a few or a lot of iterations. We can take sampling according to the loop invocations. Fig. 9 shows the precision and the efficiency of such sampling techniques. The result shows that the data dependence profiling for loops can have a very high precision even at the rate of 1/256. The overhead can be reduced to below15% of the original execution time. We can also sample the loop based on loop iterations, i.e. switch between two program versions after a predetermined number of loop iterations. This will incur more overhead than that in invocation-based techniques because of more frequent switching between two versions. However, this technique may be more appropriate for outer loops that are invoked less frequently than the inner loops.

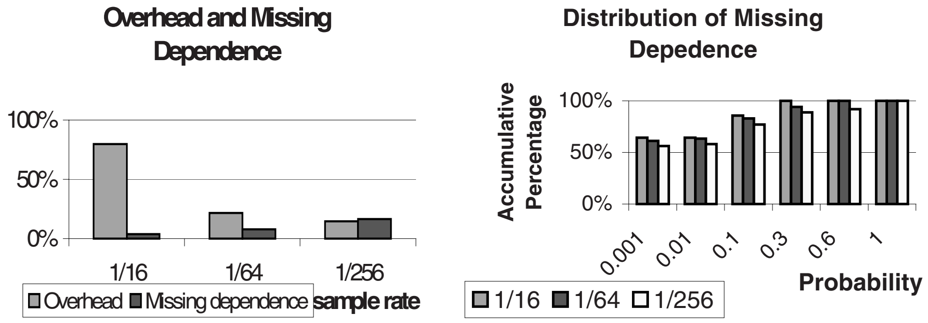

Fig. 9. Loop-oriented sampling

5 Related Work

Dependence profiling is used in [23] to help the optimizations for hyper-blocks. The profiling is supported by a special hardware structure called the *conflict buffer*. The scheme is quite efficient, compared with our instrumentation-based approach. However, loop structures are not considered in their work. The detailed and useful information about dependences, such as distance vectors and probability, is not generated. The static analysis to gain probability information has been tried in [26], but is limited to array references with linear subscript expressions only. Recent work done by [27] also uses hardware supported profiling to help thread generation. The profiling result is again limited, compared to our approach. Because of the cost of dependence profiling, a co-processor is used to improve the efficiency in [13]. Our instrumentation-based approach, though slower than hardware approach, is a general approach. It can be used when hardware support is not available, and it is more flexible. It is also easier to integrate the compile-time information with the runtime information. With the sampling techniques described in section 4, the overhead of our approach can be reasonably low.

Alias profiling is another type of profiling for memory disambiguation [21]. The dependence profiling is able to provide more accurate information than the alias profiling because the alias profiling is limited by its naming scheme. However, using the finite naming space makes the alias profiling more efficient. The dependence profile can be directly fed into the dependence graph in the compiler. Both dependence profile and alias profile can be fed into the SSA form in the compiler. However, alias profile matches the location factor scheme while the dependence profile is close to statement factor representation.

6 Conclusions

We present a tool for data dependence profiling. This tool is able to produce detailed data dependence information for nested loops and summarized dependence information for function calls. Data dependence profiles can be fed back to compiler to support speculative optimizations.

We observe that the data dependence profiling could be quite helpful in addition to static compiler analysis. Our tool identified a large amount of dependence edges in SPEC CPU2000 benchmarks with a low probability. This information is difficult to obtain using only static compiler analyses. Such information could be very useful for data speculation in optimizations such as PRE and code scheduling.

We also present several schemes to improve the timing and the spatial efficiency of the profiling tool. We use a shadow space with a simple hashing scheme to facilitate fast address comparison for detecting data dependences. It is very important to select appropriate data granularity and the size of shadow entries to minimize the total size of the shadow space.

With our proposed enhancement, we believe data dependence profiling can be very useful for compilers that support data speculative optimizations.

Acknowledgements. This work is supported in part by the U.S. National Science Foundation under grants EIA-9971666, CCR-0105574 and EIA-0220021 and grants from Intel and Unisys.

References

[1] U. Banerjee, Dependence Analysis for Supercomputing. Kluwer Academic Publishers, 1988.

[2] Michael Wolfe. Optimizing Supercompilers for Supercomputers. MIT Press, Cambridge MA, 1989.

[3] Pugh, W.: The Omega test: a fast and practical integer programming algorithm for dependence analysis_; Communications of the ACM, 35, 8 (1992), 102-114.

[4] Richard L. Kennell and Rudolf Eigenmann. Automatic Parallelization of C by Means of Language Transcription, Proc. of the 11th Int'l Workshop on Languages and Compilers for Parallel Computing, 1998, pages 157--173. Lecture Notes in Computer Science, 1656, pages 166-180.

[5] L. J. Hendren, J. Hummel, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation, pages 218--229, June 1994.

[6] R. Ghiya and L.J. Hendren. Is it a tree, a dag, or a cyclic graph? In Symposium on Principle. of Program. Language, New York, NY, January 1996. ACM

[7] M. Hind. Pointer analysis: Haven't we solved this problem yet? ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pages 54-61, Snowbird, Utah, June 2001.

[8] Rakesh Ghiya, Daniel Lavery and David Sehr. On the Importance of Points-To Analysis and Other Memory Disambiguation methods For C programs. In Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation, page 47-58, June 2001.

[9] C. Dulong. The IA-64 Architecture at Work, IEEE Computer, Vol. 31, No. 7, pages 24-32, July 1998.

[10] G.S. Sohi, S.E. Breach, T. N. Vijaykumar. Multiscalar Processors. The 22nd Annual International Symposium on Computer Architecture, pp. 414-425, June 1995.

[11] J.-Y. Tsai, J. Huang, C. Amlo, D.J. Lilja, and P.-C. Yew. The Superthreaded Processor Architecture. IEEE Transactions on Computers, Special Issue on Multithreaded Architectures, 48(9), September 1999.

[12] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, and Wen-mei W. Hwu, "Dynamic Memory Disambiguation Using the Memory Conflict Buffer," Proc. Sixth Int'l Conf. on ASPLOS, October 1994, pp. 183-193.

[13] C. Zilles and G. Sohi. A programmable co-processor for profiling. In Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7), January 2001. 12.

[14] R. Kennedy, S. Chan, S. Liu, R. Lo, P. Tu, and F. Chow. Partial Redundancy Elimination in SSA Form. ACM Trans. on Programming Languages and systems, v.21 n.3, pages 627-676, May 1999.

[15] R. D.-C. Ju, S. Chan, and C. Wu. Open Research Compiler (ORC) for the Itanium Processor Family. Tutorial presented at Micro 34, 2001.

[16] R. D.-C. Ju, S. Chan, F. Chow, and X. Feng. Open Research Compiler (ORC): Beyond Version 1.0, Tutorial presented at PACT 2002

[17] Matthew Arnold and Barbara G. Ryder A Framework for Reducing the Cost of Instrumented Code SIGPLAN Conference on Programming Language Design and Implementation, 2001.

[18] http://www.specbench.org/osg/cpu2000/

[19] R. Kennedy, S. Chan, S.-M. Liu, R. Io, P. Tu, and F. Chow. Partial redundancy elimination in SSA form. ACM Trans. Program. Languages and Systems, May 1999.

[20] J. Lin, T. Chen, W.C. Hsu, P.C. Yew, Speculative Register Promotion Using Advanced Load Address Table (ALAT), In Proceedings of First Annual IEEE/ACM International Symposium on Code Generation and Optimization, pages 125-134, San Francisco, California, March 2003

[21] Jin Lin, Tong Chen, Wei-Chung Hsu, and Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook. Ngai, Sun Chan, A Compiler Framework for Speculative Analysis and Optimizations, Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2003

[22] pfmon project web site: http://www.hpl.hp.com/research/linux/perfmon/

[23] D. Callahan, J. Cocke, and K. Kennedy. Estimating Interlock and Improving Balance for Pipelined Architectures. Journal of Parallel and Distributed Computing, 5:334-348, 1988.

[24] James Larus. Whole program paths. In Programming Languages Design and Implementation (PLDI), 1999.

[25] Thomas Ball, Peter Mataga, and Mooly Sagiv. Edge profiling versus path profiling: The showdown. In Conference Record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 1998.

[26] R. D.-C. Ju, J. Collard, and K. Oukbir. Probabilistic Memory Disambiguation and its Application to Data Speculation, Computer Architecture News, Vol. 27, No.1, March 1999.

[27] Michael Chen and Kunle Olukotun. TEST: A Tracer for Extracting Speculative Threads. In Proceedings of First Annual IEEE/ACM International Symposium on Code Generation and Optimization, pages 301-312, San Francisco, California, March 2003