Alias and dependence profiling in ORC and their applications

Tong Chen, Chu-Cheow Lim, Tin-fook Ngai and Roy Ju
MRL Intel

1. Introduction

This extended abstract describes a profiling tool to complement the static memory disambiguation analyses done normally in compilers. This technique is practical and shown to be useful in various compiler optimizations.

The abstract is organized as follows. This section first describes the motivation of this tool. Section 2 then goes into what each component of the tool does. The experimental results are reported in section 3. We then describe some current applications of this tool in Section 4, and conclude in section 5.

Memory disambiguation is one of the most important analysis techniques in an optimization compiler. Static analyses often fail to give good memory disambiguation because of complications in C or C++ programs, e.g. arbitrary type casting and arithmetic, the anonymity of heap objects, nested data structures, subscripted subscript or a combination of cases. Many state-of-art analysis methods have been proposed to attack these problems, but these are conservative.

Another inherent drawback of static analyses is that it cannot describe the memory disambiguation in the runtime behavior of a program. Profiling is a promising approach to produce information that closely matches to a program's commonly observed runtime behavior. Like profiling on control flow, we now take a profiling approach to memory disambiguation as well.

The Itanium family of processors supports data speculation via an Advanced Load Address Table feature. This feature provides the hardware support to use memory disambiguation profiling in optimizations. Furthermore, profiling can also be used to measure the effectiveness of static analysis and optimization heuristics.

In the same vein as the compiler analyses, our memory disambiguation profiling tool has two components: alias profiling and data dependence (DD) profiling. Unlike the compile analyses, the two profiling components are totally independent. The alias profiling is intended to produce target sets for indirect references, while the DD profiling is intended to produce the set of dependence edges among references. It is quite interesting to compare the two profiling methods.

Alias profiling uses a finite name set to present the memory objects in a program and collect the targets in terms of the names for indirect references. Our profiling supports different naming schemes. Whether two references are overlapped is determined by comparing their target sets after profiling.

DD profiling compares references by their addresses, and therefore usually may be more precise due to the smaller granularity in the comparison. DD profiling however appears to be more expensive.

The consumers of the two profiling methods may be different too. Alias profiling results can be easily fed to the target set of a reference during compilation, and then used in a data flow analysis and data dependence analysis. The results of DD profiling are more suitable to be fed back to data dependence graph. Since alias profiling does not take loop structures into consideration, DD profiling is able to provide more precise memory disambiguation information for loop-oriented optimizations. So we provide the two profiling methods for users to select.

2. Memory disambiguation Profiling

2.1 Overview

Our memory disambiguation profiling tool is instrumentation based. The tool is independent of hardware platforms. The result can be easily fed back to compiler for optimizations on different levels of IR. The tool is designed for offline use.

There are three phases in our profiling tool: instrumentation phase, profiling phase and feedback phase. First, the instrumentation phase inserts function calls to the profiling library routines. Next profiling is performed for a typical run of the program. Finally, the profiling result is fed back to compilers.

We build our instrumentation and feedback components in Intel's Open Research Compiler (ORC). In order to provide memory profiling disambiguation for optimizations in different stages in ORC, our tool can instrument and feedback at different stages in ORC (Figure 1~~Figure 1~~). To simplify the implementation, all the instrumentation is done on the Whirl Node format of IR in ORC. During feedback, the profiling results may be propagated into other IR formats, such as hashed SSA in WOPT or operations in CG.

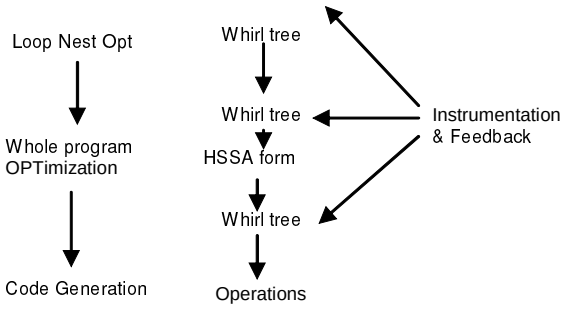


Figure 1 Overview of the profiling tool

A common feature of the memory disambiguation profiling is that the profiling tool has to compare address values or look up information according to address values of references. To speed up such operations, we provide a special hash table, called shadow, which is associated with the program space. The shadow is implemented in a similar way as paging in OS. The shadow is intended to trade space for time.

2.2 Alias profiling

Alias profiling generates the target set of each reference. The target is represented by the names of memory objects, including local variables, global variables or objects on heap. The basic idea of alias profiling is to record the name associated with each address in its shadow entry. For each reference, the targets it accesses can then be easily found out by checking its shadow entry hashed by its address value at runtime.

In the alias profiling, we instrument the following events:

- Procedure calls. The calling context is maintained.
- Allocation of global variables, local variables and heap memory objects. The global variables are allocated once in a problem. The local variables are allocated each time the procedure is called. The memory objects are allocated with system calls and are anonymous. The profiling tool assigns them names according to calling context at the allocation point. This simulates the compiler naming schemes.
- Memory references. The loads from memory or stores to memory are instrumented. Profiling will generate their target sets according to the address values found.

Alias profiling outputs the target sets for references and the read/write sets of the side effect of function calls. After the results are fed back, we can determine the alias relationship of among references or function calls by comparing their target sets or the read/write sets. While the compiler may find a pair to be possibly aliased, the profiling may reveal that it is not aliased at runtime. As such, the profiling can help the compiler to select the correct place for data speculation. Figure 3 shows how often such cases appear in all the alias queries in code generation of ORC.

2.3 Data Dependence (DD) profiling

Data dependence occurs when the two references access the same address location. It would be very costly to record all the addresses accessed by each references and compare the addresses pair-wise. We use the shadow to speed up the comparison: references accessing the same shadow entry have a dependence relation. The latest load and store references to this address are stored in the shadow entry. Therefore, dependence edges can be quickly detected.

A detected edge may be between two references in different procedures. To make the profiling results usable to compilers, we try to locate the common procedure in which the two references reside according to their calling context. The resulting dependence edges may be between references and function calls. Thus, our data dependence profiling is able to generate complete dependence graph for code with function calls.

The profiling tool also takes loops into account. The iteration vector of the loop-nest is also recorded in the shadow. When an edge is detected, the dependence distance vector can be calculated out by subtracting the iteration vector of the previous reference from the current iteration vector.

In DD profiling, the following instrumentation is done:

- Loops. The entry and exit of loops, and the beginning of iterations are instrumented so as to maintain the loop iteration vector.
- Function calls. Entry into and exit from function calls are instrumented. The calling context is maintained so that dependence of function calls can be determined.
- Memory references. The address value of memory references are used to detect data dependence.

An important aspect of DD profiling is that we can now attribute probability values to the dependence edges. The probability of a dependence edge between two references is a measure of how often the dependence actually exists during program runtime.

3. Experimental results

We use ORC 2.0 to build our instrumentation and feedback tool. Spec CPU2000 Integer benchmarks are used in the experiments. The experiments are run on a 733MHz Itanium2 workstation with 2 G memory.

First, we try to show the efficiency of the memory disambiguation profiling in Figure 2~~Figure 2~~. In this particular experiment, the instrumentation is done after WOPT, where many optimizations, such as partial redundancy elimination and register allocation, have been done to reduce the number of memory references. The DD profiling is for innermost loops here. The empirical results show that alias profiling is much faster than DD profiling.

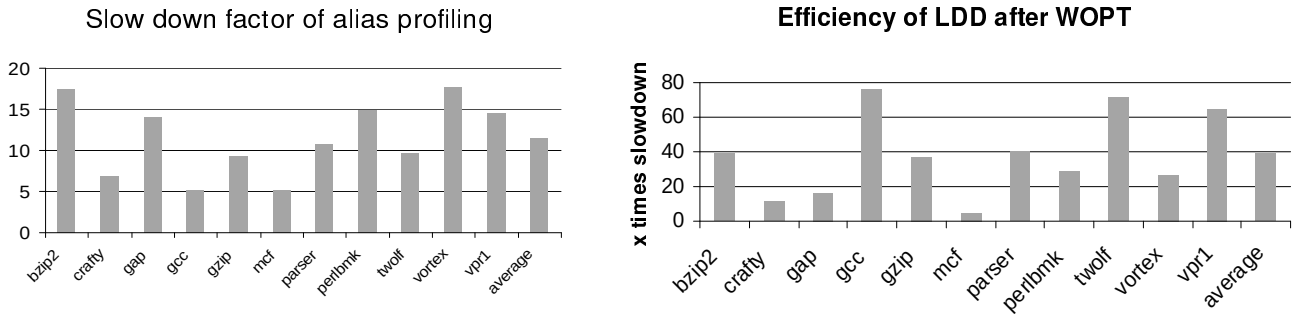


Figure 2~~2~~ Runtime efficiency of memory disambiguation profiling

To demonstrate the usefulness of the alias profiling, we try to compare the profiling result with the ORC's alias analysis result. When ORC's alias analysis says a pair of references is possibly aliased, alias profiling may say not. Figure 3~~Figure 3~~ shows the percentage of possibly aliased pairs detected by profiling to be not aliased at execution time. In this experiment, different naming schemes are also tried. G=0 means that all heap memory objects are given one name. G=h means that the heap memory objects are assigned names according to the calling context. The suffix f means that fields of structures are considered.

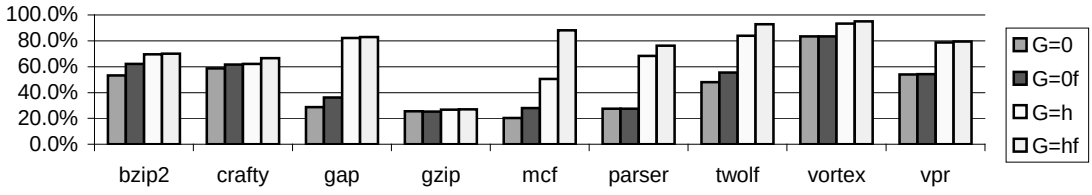
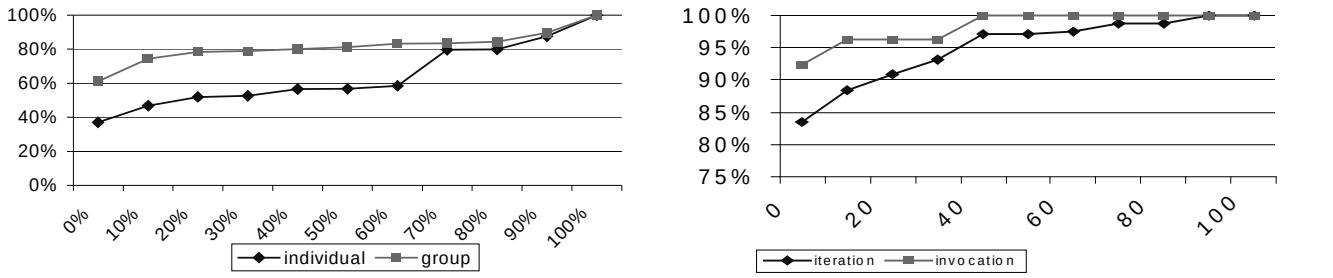


Figure 3~~3~~ Percentage of possibly aliased pairs detected as not aliased by profiling

In order to reduce the execution time of DD profiling, we applied sampling techniques. Since the goal of DD profiling is to check the correlation among references, we may lose some information when we sample the references. The sampling can be done on the references individually or in a small group at certain sample rate.

Another technique we use to reduce profiling time is loop-oriented sampling. The sampling is based on loop invocations or loop iterations. The accuracy of using sampling is shown in Figure 4~~Figure 4~~. The changes in the probability values of the edges, as a result of sampling, are calculated. These values are plotted on the x-axis. The y-axis shows the cumulative percentage of edges. A point (X, Y) indicates that Y% of the edges have a change in probability equal to or less than X. For

example, even when only a small fraction of the loop iterations are sampled, close to 85% of the edges have a 10% or less change in their probability. . We also use the predicate registers in Itanium to do the sampling quickly. Using either sampling, DD profiling time can be improved to a 7x slowdown without losing much precision. This is a significant improvement over the 40x slowdown in Figure 2.



Fraction of references sampled = 0.001 Fraction of loop iterations/invocations sampled = 0.000032 / 0.073

Figure 4 Reference-oriented sampling and loop-oriented sampling

4. Applications

4.1 Data speculative PRE

There are more opportunities for partial redundancy elimination if the data speculation is introduced. By ignoring some of the possibly aliased write, more expressions can be found to be redundant. To ensure correctness at runtime, we use the ALAT to check the speculation and recover if the speculation fails. Figure 5 shows the performance improvement of data speculative PRE guided by alias profiling and data dependence profiling respectively.

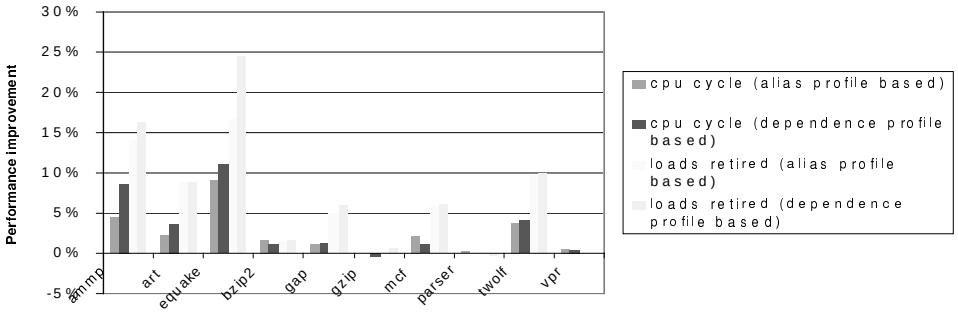


Figure 5 Performance of data speculative PRE.

4.2 Speculative parallel thread generation

Currently, the data dependence profiling is being used to guide our speculative parallel thread (SPT) generation. The dependence profiling result provides the basis for data dependence speculation. It allows the compiler to identify which critical dependences are likely and which are unlikely. Based on the dependence probabilities, our SPT compilation selects and transforms loops into optimal SPT loops. From performance results of our early experiments, we expect that with such dependence profiling feedback the compiler can generate SPT code that achieve 20+ % average speedup for Spec2000Int benchmarks.

5. Conclusions

We present a profiling tool for memory disambiguation in this paper. This tool has components for alias profiling and data dependence profiling. It uses the shadow data structure which allows the profiling efficiency to be acceptable, and makes this a practical technique. The profiling efficiency can be further improved by sampling techniques. The importance of memory disambiguation profiling is illustrated from a comparison of profiling and the static analysis results (Section 3), and the various applications to compiler optimizations (Section 4).