Compiler-Directed Static Classification of Value Locality Behavior

Qing Zhao and David J. Lilja[*]

Dept. of Computer Science and Engineering, Univ. of Minnesota, Minneapolis, MN 55455

[*]Dept. of Electrical and Computer Engineering, Univ. of Minnesota, Minneapolis, MN 55455

zhao@cs.umn.edu, [*]lilja@ece.umn.edu

Abstract

Predicting the values that are likely to be produced by instructions has been suggested as a way of increasing the instruction-level parallelism available in a wide-issue processor. One of the potential difficulties in exploiting the predictability of values, however, is selecting the proper type of predictor, such as a last-value predictor, a stride predictor, or a context-based predictor, for a given instruction. We propose a compiler-directed classification scheme that statically partitions all of the instructions in a program into several groups, each of which is associated with a specific *value predictability pattern*. This *value predictability pattern* is encoded into the instructions to identify the type of value predictor that will be best suited for predicting the values that are likely to be produced by each instruction at run-time. Both an idealized profile-based compiler implementation and an implementation based on the GCC compiler are studied to show the performance bounds for the proposed technique. Our simulations based on the SimpleScalar tool set and the SPEC95 integer benchmarks indicate that this approach can substantially reduce the number of read/write ports needed in the value predictor for a given level of performance. This static partitioning approach also produces better performance than a dynamically partitioned approach for a given hardware configuration. Finally, this work demonstrates the connection between value locality behavior and source-level program structures thereby leading to a deeper understanding of the causes of this behavior.

1 Introduction

Several constraints limit the instruction-level parallelism (ILP) that can be exploited in programs, including true data dependencies (read-after-write hazards), artificial dependencies (write-after-read and write-after-write hazards), control dependencies (branches), and resource conflicts (such as limited numbers of register ports or function units). While the performance impact of control dependencies and artificial dependencies can be reduced or sometimes even eliminated using various hardware and software techniques, such as branch prediction and register renaming, true data dependencies remain a serious limitation in exploiting ILP in superscalar processors. Value prediction has been proposed as a mechanism to break true data dependencies [2,3, 6-15].

Recent studies have shown that wide-issue processors benefit more from value prediction than traditional four-instruction-issue superscalar processors since more true data dependencies typically appear in the larger instruction windows needed to support wider issue widths [2]. Eliminating data dependencies in these larger instruction windows potentially can increase performance substantially. On the other hand, several problems arise when applying value prediction in wide-issue processors. For instance, since multiple instructions need to access the value predictor in each cycle, the predictors must be able to support very high read/write bandwidths. Our measurements on a set of SPEC95 integer programs, which are summarized in Table 1, show that a simple value predictor (the SVP described in Section 4.3) must support an average of 5 accesses per cycle for an 8-issue processor, increasing to 7 accesses per cycle for a 16-issue processor. If the value predictor is unable to support this request bandwidth, many predictions that could otherwise have been made will not be available to the instruction issue logic. Furthermore, entries in the value predictor will not be updated with the newly generated values in time for the next prediction request, which will cause a corresponding increase in the misprediction rate.

issue width	benchmarks								
	go	m88ksim	gcc	compress	li	ijpeg	perl	vortex	AVG
8_issue	5.2	5.5	4.6	5.2	4.5	6.1	4.7	4.2	5.0
16_issue	6.1	10.7	6.3	6.7	6.3	11.0	6.1	6.0	7.0

Table 1. The average number of requests per cycle made to a simple value predictor in a wide-issue superscalar processor.

There are several approaches that can be used to resolve this value prediction bandwidth problem. A straightforward approach is to increase the number of read/write ports in the value predictor. While multiple independent ports will increase the number of requests per cycle that can be handled by the predictor, each additional port increases the complexity of the design with additional decoding and multiplexing logic and, perhaps more importantly, it substantially increases the wiring density. This additional circuitry and increased wire density will increase the implementation cost. Furthermore, these changes most likely increase the critical path delays, which then will increase the predictor's access time and the processor's clock period. A variation of this approach interleaves the predictor using several banks with a fast distribution network [2]. This scheme still suffers from bank conflicts, however, since several instructions may try to access the same bank simultaneously. Moreover, this interleaving requires a scaling factor that is a power of two, which severely restricts the development of a cost-effective design for value predictors in high-issue width processors [4].

A decoupled value predictor scheme has been proposed [3] that reduces the bandwidth requirement of the predictor by dynamically classifying each instruction into one of several different component predictors. One drawback of this scheme, though, is that it requires a modified trace cache to store the classification information for each instruction. Additionally, this dynamic classification scheme suffers from a relatively long latency required to classify instructions. Due to changes in the predictability behavior of some instructions during a program's execution, these instructions migrate among the several different component predictors at run-time. These frequent migrations will introduce many transient states into the predictors, thereby reducing their effectiveness.

This paper proposes a new compiler-directed classification scheme for statically partitioning individual instructions among several different component value predictors. The compiler attaches information about the *value predictability pattern* for each instruction to its binary object code through a simple extension of the instruction format. This compiler-directed static classification scheme reduces the hardware cost and complexity required to build a large prediction cache with multiple ports. Instead, several smaller value predictors with only a limited number of ports in each can provide the equivalent bandwidth of a single larger multiported predictor while maintaining approximately the same, and in some cases even slightly better, prediction accuracy. Since each instruction is statically marked with its prediction classification, no additional hardware classification table is needed. This preclassification also eliminates the time required by existing dynamic classification schemes to classify the value predictability behavior of instructions and it eliminates the start-up overhead incurred when instructions migrate among several different component predictors. It also eliminates the need for a single instruction to occupy multiple entries simultaneously in the component predictors of the overall predictor.

Unlike other compiler-directed techniques that rely on profile-guided heuristics [5], our static classification scheme identifies the salient characteristics of predictable instructions at compile time without the need to profile the program beforehand.

The remainder of this paper is organized as follows: Section 2 summarizes recent related work on value prediction schemes. Section 3 then studies the inherent characteristics of programs that serve as the basis for the static classification. This section also describes the compiler classification heuristics and presents classification results for an idealized compiler. Our performance evaluation methodology is described in Section 4 with the simulation results presented in Section 5. Finally, the results and conclusions are summarized in Section 6.

2 Related Work

There are three main areas of previous work related to the scheme proposed in this paper. The first area focuses on building more accurate and efficient value predictors, such as the last-value predictor [6], the stride value predictor [7], and the context-based value predictor [8,9]. Each of these predictors produces good performance for certain value locality patterns, but poor performance for others. To obtain higher prediction rates for instructions with varying value locality patterns, several simple value predictors can be combined into one hybrid value predictor [9,10,11]. Although these hybrid value predictors can provide higher prediction rate than single predictors, they can waste resources since every instruction being predicted occupies a unique entry in each of the component predictors. To address this inefficient use of the predictor resources, dynamic classification schemes [3,10,11] can distribute the instructions into the different component predictors at run-time.

Another important area of related work has studied how to incorporate value predictors in complete machine models. Gabbay and Mendelson [2] showed that the performance potential of value prediction tends to increase as the instruction issue width increases. However, it has been pointed out [2,3] that using value prediction in wide-issue processors requires very high bandwidth predictors. To address this problem, a highly interleaved prediction table has been proposed [2]. Additionally, Lee *et al* [3] combined a dynamic classification table with a trace cache to distribute instructions to different component predictors thereby increasing the predictor bandwidth available in each cycle. Another approach is to reduce the bandwidth requirement of the predictors by giving priority to those instructions that belong to the longest data dependence chains [12]. Rychlic *et al* [11] have shown that over 30% of the value predictions in the SPEC95 integer programs produced no performance improvement, which further supports the selective prediction scheme.

Finally, the last area of important related work has studied the program characteristics that lead to value locality [14,15]. For instance, Sazeides and Smith [14] divided prediction models into two groups, computational and context-based, while Sodani and Sohi [15] studied the sources of instruction repetition at both the global level and the local level.

3. Static Classification

A program's execution-time behavior determines the predictability of the values its instructions produce. There are two basic types of value predictability in programs [14]:

- *Value repetition*, in which subsequent dynamic instances of the same static instruction repeat the values produced by previous dynamic instances. The *repetition distance* is the number of times an instruction is executed between two repetitions of the same value (including the repeating instance itself). For instance, an instruction that always produces the same value has a *repetition distance* of one, while an instruction that alternately produces only two distinct values has a repetition distance of two. Note that the *repetition distance* need not be constant for a particular static instruction.
- *Value computability*, in which values produced by later dynamic instances of the same static instruction can be computed using a simple function of the previous values.

Last-value predictors [6] and context-based value predictors [8,9] have been proposed to capture the value repetition that exists in programs. The primary difference between these two types of predictors is the *repetition distance* that can be exploited by each. The last-value predictors assume that the immediately previous value produced by an instruction will be produced again the next time it is executed. Thus, it assumes a *repetition distance* of one. The context-based predictors, on the other hand, generalize to arbitrary repetition distances by selecting one of several previous values as the next value predicted to be produced by an instruction. Computational predictors, such as the stride value predictor [7], compute the next predicted value using some predefined function of previously seen values, such as an increment by a constant stride.

In this section, we further categorize the predictability behavior of instructions with the goal of developing compiler algorithms to statically identify the *value predictability pattern* of each instruction.

3.1 Value Predictability Patterns for Static Instructions

We classify static instructions into three different groups according to the predictability behavior that they exhibit. This classification was developed by observing both the run-time behavior of programs and the source code level structures that can be naturally analyzed by compilers.

- Single_Pattern -- These instructions have exactly one type of value predictability during the execution of the program. For example, the sequence {1, 2, 3, 4, 5, 6, ...} demonstrates a computable value predictability pattern while the sequence {4, 7, 9, 4, 7, 9, ...} shows a single type of value repetition. We further subdivide this group below.

- Mixed_Pattern -- Instructions with this category may interleave both the value repetition and the value computability types of behavior. The value sequence {1, 2, 3, 5, 5, 5, 5, 1, 2, 3, 5, 5, ...}, for instance, shows both repetition of values and a stride-based computable pattern.
- Unknown -- This group includes static instructions whose values are not predictable during the program execution.

It is useful to further divide static instructions in the Single_Pattern group into the following three subgroups:

- The Single_Last subgroup includes those instructions that have a constant *repetition distance* of one. That is, the instruction usually produces the same value as the immediately previous instance.
- The Single_Stride subgroup includes instructions whose values are computable using a constant stride.
- The Single_Context subgroup includes instructions with a value *repetition distance* larger than one. For example, the dynamic instruction instances sequence {1, 4, 17, 23, 1, 4, 17, 23, 1, ...} exhibits value repetition with a *repetition distance* of four. The corresponding static instruction thus belongs to this Single_Context subgroup.

Note that static instructions in both the Single_Last and Single_Context subgroups show similar value repetition behavior, differing only in the *repetition distance*. Logically, they could be combined into a single group with an associated value *repetition distance*. However, the Single_Last type is pulled out as a separate subgroup because instructions with this behavior occur quite frequently in the programs tested.

Each group and subgroup corresponds to a particular type of program structure. Instructions in the Single_Last group typically are assigned a value at the beginning of a loop's execution and then are never changed within the loop. These instructions are said to be part of *loop invariant computations*. On the other hand, a *loop induction variable* that is a variable whose successive values form an arithmetic progression in a loop [17], produce the Single_Stride behavior. An instruction within the innermost loop of a set of nested loop could produce the Single_Context behavior when the inner loop causes the instruction to produce some arbitrary sequence of values and the outer loop causes this sequence to be repeated. The *repetition distance* for the values produced by this instruction will be the number of iterations in the innermost loop.

While the above simple program features cause the Single_Pattern behaviors, program structures that cause the Mixed_Pattern behaviors are more complicated. Figure 1 shows a fragment of a small program that illustrates one type of situation in which the Mixed_Pattern behaviors could occur. In the two-level nested loop, there are three static instructions *I1,I2,I3,* that produce different *value predictability patterns*, as noted in the comments in this figure. Instruction *I1* cannot be moved outside of the loops by an ordinary loop invariant removal compiler optimization because *I1* does not dominate the use of variable *r1* in *I3* [17]. It is obvious that this instruction will produce only a single value throughout the loop's execution. Instruction *I2* contains an induction variable on the inner loop and so demonstrates a Single_Stride pattern. Note that the operand *r1* of instruction *I3* has two possible definitions, one defined by *I1* and the other by *I2*. Since these definitions occur in complementary control paths that both converge at *I3*, the value predictability patterns of *I1* and *I2* will determine the value predictability pattern of *I3*. As a result, *I3* shows a Mixed_Pattern behavior. Specifically, it shows the value repetition behavior with a *repetition distance* of one in the first iteration of the inner loop while showing the value computability behavior in the subsequent iterations of the inner loop.

Instructions in single-level loops that perform complex computations will produce an arbitrary sequence of values during execution. For example, traversing a linked list would often generate address values that have no value predictability. Instructions in this type of program structure would demonstrate an Unknown value predictability pattern.

```
for ( flag = 1; flag <= N; flag++ ) {
   for ( i = 0; i <= M; i++ ) {

         if (flag == 1) {
/*I1*/        r1 = 4;       /* Single_Last */
         }
         else {
/* I2*/        r1 = i + 2; /* Single_Stride */

                            }
/* I3*/  r2 = r1 + 4;       /* Mixed_Last_Stride */
   }
}
```

Figure 1 An example of a program structure that causes a Mixed_Pattern value repetition behavior.

3.2 Compiler Classification

Now that we have identified an appropriate set of groups with which to classify the value predictability of instructions, we develop compiler algorithms to classify each static instruction in a program. We begin by describing an idealized compiler algorithm that uses the execution profile of a program to classify the instructions. This idealized compiler is used to provide an upper bound on the performance potential of this classification approach. We then describe our implementation of an approximate classification in the GCC compiler.

3.2.1 Idealized compiler

The classification information for the idealized compiler is collected by a profiler as the program is executed. The profiler stores in execution order the output values produced by up to 100 consecutive instances of each static instruction. Whenever a new instance of an instruction is encountered, the value repetition behavior (with a *repetition distance* up to 100) and the computable stride are checked simultaneously using three counters. One counter is used to determine value repetition behavior with a repetition distance of one, the second counter tracks repetition distances larger than one, and the third is used to determine if there is a computable stride.

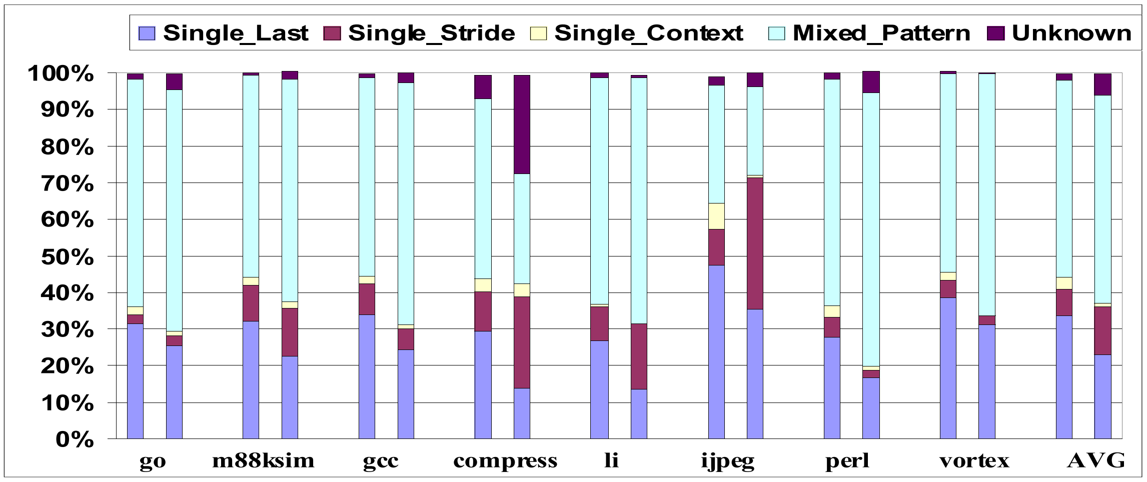


Figure 2. The distribution of different value predictability patterns when using the idealized profile-based value predictability classification algorithm. The left-hand bar for each program corresponds to the static instruction count and the right-hand bar corresponds to the dynamic instruction count.

At the end of a program's execution, if only the counter for a repetition distance of one is non-zero, the instruction is assigned to the Single_Last group. Similarly, if only the stride counter is non-zero, or if only the counter for repetition distances larger than one is non-zero, the instruction is assigned to the Single_Stride group or the Single_Context group, respectively. Otherwise, if two or three different counters are non-zero, the instruction is assigned to the Mixed_Pattern group. Finally, if all of these counters are zero, the instruction is assigned to the Unknown group.

Figure 2 shows the distribution of instructions executed by the test programs when divided into the above five value predictability patterns using the idealized profile-based compiler algorithm. From these results, we can see that, on average, the Single_Last and Single_Stride categories together comprise about 40% of both the static and dynamic instruction counts. An average of 50-60% of both the static and dynamic instruction counts are classified as showing the Mixed_Pattern value predictability behavior. Typically, no more than about 5-10% of the instructions are classified as either Single_Context or Unknown.

3.2.2 Practical Compiler Heuristic

The statistics in Figure 2 suggest that a practical compiler algorithm to perform this static classification of value predictability behavior can ignore the instructions in the Unknown group and the Single_Context subgroup since they occur relatively infrequently. The task then becomes to partition the whole program into three categories: Single_Last, Single_Stride and Mixed_Pattern. In fact, we develop a compiler algorithm to classify instructions into both the Single_Last and Single_Stride categories. Every instruction that does not get classified in either of these two categories must be either Mixed_Pattern, Unknown, or Single_Context instructions. Since the fraction of instructions that the profile-based algorithm determined were either Unknown or Single_Context is quite small, we simply combine instructions in these two categories with the Mixed_Pattern instructions to make a single category called **Others**.

In this paper, we describe an approximate static classification algorithm that we implemented in the GCC compiler, version 2.6.3. This algorithm is implemented using only the information available in a single function at-a-time since this is the most basic level for practical compilers. This algorithm could be implemented in any compiler, even if it does not have interprocedural analysis, accurate alias analysis and other modern compiler control flow and data flow analysis algorithms. Therefore, this function-level approximate algorithm provides a lower bound on what could actually be achieved in a real compiler on a real system.

As described in Section 3.1, loops are the basic source-level program structures that lead to the Single_Last and Single_Stride value predictability behavior. Consequently, identifying instructions that fall into the Single_Last and Single_Stride categories can be accomplished by using the compiler to identify *loop invariant computations* and *loop induction computations*, respectively.

Many loop invariant computations can be eliminated by a traditional loop invariant removal compiler optimizations. This optimization tries to identify computations that never change within a loop and then moves them outside of the loop body. There are several compile-time limitations that can hinder this optimization, however. For example, if the compiler has no information about the life span of a particular result produced within a loop, it will be unable to determine the benefit of removing the computation from the loop. In this case, it simply may leave the computation within the loop. In GCC, for instance, a computation is moved out of a loop only once. As a result, a computation that has already been moved out of the inner loop will not be moved again outside of an outer enclosing loop, even though it may be an invariant within the outer loop as well.

Another situation in which a compiler typically cannot move a loop invariant computation out of the loop body is when the loop invariant computation does not dominate (in a graph-theoretic sense) all of the uses of it within the loop and all the exits of the loop [17]. The compiler cannot move the computation out of the loop in this case since it cannot prove that correct program operation will be maintained.

A third situation that limits the compiler's ability to move loop invariant computations outside the loop is due to function boundaries. Although functions can be called within loops, a function-level compiler has no interprocedural information since it analyzes only one function at a time. As a result, a function level compiler ignores all the loop invariant computations that occur within called functions. In Figure 3, for example, the function *abc* is called by the function *foo* in a *for* loop. Instruction *I1* in function *abc* obviously is a loop invariant computation within the loop in the function *foo.* However, when the function level compiler analyzes the function *abc*, instruction *I1* does appear not within a loop, so it is not identified as a loop invariant computation. As a result, no loop invariant removal optimization is applied on instruction *I1*. Note that interprocedural analysis cannot always solve this problem if a function is called from multiple sites. Function in-lining can be used to make instruction *I1* visible as a loop invariant computation in this example, although in-lining can introduce other difficulties.

The above three situations at least partially explain why the Single_Last value repetition behavior still exists in programs at run-time even after the compiler has aggressively tried to remove loop invariant computations. Similarly, the Single_Stride behavior still exists at run-time even though there are compiler algorithms to remove loop induction computations [17]. The basic optimization is to convert the arithmetic progression produced by a simple loop induction computation into a corresponding closedform function of the loop index. However, the values of the induction variables themselves are typically used within the loop iterations. Consequently, most of the loop induction computations are retained within the loop after the compiler has completed its optimizations.

```
foo() {                                          abc() {
      …                                              int r1;
      for (i = 0; i<= N; i++)            /*I1*/      r1 = 6;        /* loop invariant*/
          abc();                                     …
}                                                }
```

Figure 3. An example of a loop invariant computation that is not visible to a compiler that has no interprocedural analysis capabilities.

Based on the above analysis, the identification of instructions that belong to the Single_Last and Single_Stride categories can be divided into two steps in the function level compiler. The first step is to apply the traditional loop invariant and loop induction computation identification algorithm after the normal loop optimization pass. These algorithms will find the Single_Last and Single_Stride instructions within the loops of the current function being analyzed. Next, we identify *potential* Single_Last and Single_Stride instructions by assuming that every function analyzed is called within a loop. This allows us to put instructions in the Single_Last group which would otherwise not be identified by a traditional compiler as being loop invariants.

Note that our classification of instructions is used only to assist a later value prediction at run-time and not to produce any program transformations that must be guaranteed to be correct to maintain the semantics of the program. As a result, if our assumption is wrong for some functions, that is, if the functions are not called within loops, the computation still will be correct. The worst that will happen is that the value prediction performed at run-time will be incorrect. This wrong prediction may impact performance, but not program correctness. Furthermore, notice that, if the function is not called within a loop, it will be executed a relatively small number of times. As a result, the computation that our algorithm identified as being potentially loop invariant will be executed only a relatively small number of times. Thus, misclassifying the instructions in this function as Single_Last will have very little impact on the overall classification.

The algorithm to identify the *potentia*l loop invariant computations in the second step above requires only simple function-level data-flow analysis. First, we define the *original data source* to be the type of variable that provides the input data for a function. We identify these *original data sources* to be immediate values (CONST), function parameters (ARG), returned values (RET), global values (GLOBAL) and stack pointer related values (SP). An *original definition* is a definition of the value of a variable within a function that has one of these *original data sources* on the right side of an assignment statement. Thus, the type for an *original definition* is one of CONST, ARG, RET, GLOBAL and SP. The data-flow analysis begins by finding all the *original definitions* in a function that have specific original definition types. The compiler then applies traditional def-use data-flow analysis on the entire function to associate all the *original definitions* with each instruction. Finally, each instructions is categorized as Single_Last if all of its *original definitions* are of the CONST type, even if they are not within loops in the current function.

In addition to using the above algorithms to identify instructions in the Single_Last category, we can identify some Single_Last instructions using specific features of the instruction set [15]. For example, when the number of bits in the immediate field of an instruction format limits the size of the immediate value that can be handled by an instruction, larger constants often are manipulated using a standard sequence of instructions. In the MIPS architecture, for instance, the "lui" instruction commonly is used for generating this type of large immediate value. We take advantage of these instruction set architecture-specific behaviors by using the op-code generated by the compiler to identify these types of instructions as belonging to the Single_Last category. In this study, in particular, we optimistically classify all "lui" instructions as Single_Last.

3.3 Architectural support

To pass the information acquired by the compiler to the processor, we add a three-bit extension to each instruction. These extra three bits identify one of the five predictability patterns identified by the idealized profiler-based compiler (Single_Last, Single_Context, Single_Stride, Mixed_Pattern, Unknown), or one of the three predictability patterns identified by the real compiler (Single_Last, Single_Stride and **Others**).

4. Performance Evaluation

We use the above compiler-directed classification algorithm to statically assign instructions to one of three different component predictors within an overall value predictor. This combined predictor is used within a wide-issue superscalar processor to predict the values that will be produced by individual instructions. We evaluate the performance of this statically-classified predictor using both the idealized profile-based algorithm and the function-level algorithm implemented in the GCC compiler. These two compilers provide lower and upper bounds on the performance that could be expected from this type of static-classification predictors. We compare our results to both a simple hybrid value predictor and a predictor that dynamically distributes instructions to the component predictors using only run-time information.

4.1 Simulation methodology

We have developed a cycle-accurate execution-driven simulator derived from the sim-outorder simulator in the SimpleScalar tool set [1] for this study. The baseline superscalar processor supports out-of-order instruction issue and execution. To prevent resource and control dependences from becoming performance bottlenecks, and instead allowing us to focus on the performance potential of the value predictors themselves, we assume perfect branch prediction and a sufficient number of function units. The processor can issue 16 instructions per cycle out of a 256-entry instruction window, and the load-store queue has 64 entries.

There are many ways to incorporate the value predictors into a superscalar processor and it is beyond the scope of this paper to propose an optimal design. In this study, we adopt a conventional approach for integrating the value predictors into the processor pipelines. The value predictors are indexed by the instruction addresses during the instruction fetch stage. The predicted values then are inserted into the reservation stations of the data-dependent instructions after the instructions are decoded in the dispatch stage. These instructions in the reservation stations are marked as having one or more predicted input values. An instruction in the reorder buffer will be scheduled and executed when all of its operands are available and there is an available functional unit, regardless of whether or not the operands are predicted. When the actual results are produced, the predicted value is verified and updated during the write-back stage. If the actual result is the same as the predicted result, the execution is continued without any interruption. Otherwise, only those instructions that were issued using this incorrectly predicted value will be invalidated and selectively reissued with the correct value [11]. Values are predicted only for load instructions and for register instructions whose outputs are needed by other instructions in the current instruction window [3].

4.2 Benchmark programs

Benchmark	Input Set	No. of Inst. Executed (million)	No. of Inst. need prediction (million)	IPC for baseline machine	Full Speedup over baseline machine(%)
go	5,9	82	63	7.29	8.0
m88ksim	dcrand.lit	240	164	9.34	68.0
gcc	jump.i	38	25	8.54	14.4
compress	10000 e 2231	35	23	8.70	17.1
li	queen6.lsp	41	24	9.98	8.0
ijpeg	spectriv.ppm	98	76	10.61	33.2
perl	scrabbl.in	40	25	7.50	30.6
vortex	persons.250	66	39	8.67	18.0

Table 2. The run-time characteristics of the SPEC95 integer benchmark programs used in this study

We use eight integer programs from the SPEC95 benchmark suite for this performance evaluation. The inputs sets used, the total instruction counts, and the IPC of the baseline processors are shown in Table 2 for these test programs. The fourth column in this table shows the total number of instructions that need to query the value predictor. The last column shows the speedup obtained compared to the baseline processor when using a simple value predictor (the SVP described in Section 4.3). This comparison assumes that the value predictor always has a sufficient number of read/write ports available so there are never any conflicts among instructions trying to access the predictor at the same time. All the programs were compiled using our modified GCC 2.6.3 compiler with –O3 optimization and loop unrolling enabled. A few of the input sets were modified slightly to control the simulation time.

4.3 The comparison value predictors

We compare our Static Classification Value Predictor (SCVP) to two other value predictors. The first one is the Simple Classification Value Predictor (SVP). This hybrid predictor consists of three component predictors, a last-value predictor, a stride predictor and a context-based predictor. Every predicted instruction is allocated an entry in each of the three component predictors. When a value is to be predicted, each component predictor is queried. We assume an idealized predictor such that the correct prediction from the three component predictors is always selected, if the corresponding 3-bit confidence counter for the corresponding component predictor exceeds the predefined prediction threshold value. This predictor can still mispredict when all of its component predictors mispredict. Furthermore, no prediction will be made when the confidence counters of all of its component predictors are below the prediction threshold, or if the instruction is not in the prediction table and so cause a table miss. This SVP serves as an ideal hybrid value predictor and is used in computing the average number of requests in Table 1 and the ideal speedups shown in Table 2.

The second predictor used in our comparisons is the Dynamic Classification Value Predictor (DCVP) [3]. This predictor uses a classification table to dynamically assign instructions into one of the three component predictors of the above SVP. The first time an instruction whose output needs value prediction is executed, it is assigned the Unknown type and forwarded to the classification table. The output value is stored in the classification table at this time. When it is executed the second time, a stride value is calculated by subtracting the old output value stored in the table from the new output value. The stride value calculated and the new output value then are stored in the table. If this instruction is executed again, a new stride value is calculated and compared with the old one stored in the table. If both of them are equal to zero, this instruction is assigned to the Last type. Otherwise, if they are equal but are not zero, this instruction is assigned to the Stride type. Finally, if they are not equal, this instruction is assigned to the Context type.

An instruction that has been assigned an access type is dispatched to the component value predictor with the corresponding access type and will be predicted by that value predictor the next time it is executed. When the dynamic instances of this instruction cannot be correctly predicted by that value predictor several times in a row, this instruction is removed from its current value predictor, is assigned the Unknown type again, and is forward back to the classification table. After another classification stage, this instruction is assigned a new access type that may be the same as or different from the previous access type.

Since the value predictability pattern of an instruction can change from time-to-time during a program's execution, its prediction classification may change frequently in this mechanism. Frequent reclassifications will lower the performance of this predictor due to the time required to warm-up the predictor for an instruction after is has been reclassified. Note that the DCVP needs additional storage to store the classification information for each instruction and a complicated control engine to accomplish the above tasks. However, we do not include the additional storage and the overhead of this control engine in the following simulations. As a result, the simulation results for the DCVP are somewhat optimistic.

Our Static Classification Value Predictor (SCVP) used in the following experiments also includes a last-value predictor, a stride predictor and a hybrid predictor similar to the SVP. The two different levels of compiler capability that we simulate, specifically, the idealized profiler-based compiler scheme, denoted SCVP_P, and the actual GCC compiler implementation, denoted SCVP_C, use these component predictors in a slightly different way. Both compiler-based mechanisms use the last-value predictor to hold those instructions that were statically classified as showing the Single_Last value prediction behavior. Both of them also use the stride predictor for instructions in the Single_Stride category. With the idealized profiler-based compiler scheme (SCVP_P), though, the hybrid predictor is used for instructions classified as either Mixed_Pattern or Single_Context. Instructions classified as Unknown are not predicted with the SCVP_P mechanism. However, with the real SCVP_C compiler mechanism, the hybrid predictor is used to predict all instructions classified as **Others**, which includes the Mixed_Pattern, Single_Context, and Unknown categories.

SCVP					SVP			DCVP			
Last	Stride	Hybrid			Last	Stride	Context	Classify	Last	Stride	Context
		Last	Stride	Context							
4k	1k	4k	1k	8k	8k	2k	8k	4k	4k	2k	8k

Table 3. The table sizes used in the following simulations for the SCVP, SVP and DCVP value predictors

To fairly compare these different value predictors, we choose the sizes of the component predictors such that the total storage resources allocated for each of the three predictors being compared was constant. Table 3 shows the specific sizes used in each of the configurations. These table sizes for each component predictor in the classification predictors (SCVP and DCVP) were chosen based on the observed static and dynamic classification distributions (See Fig.2 and [3]).

4.4 Evaluation Metrics

The most important evaluation metric is the overall machine performance obtained when the baseline processor incorporates one of the predictors being compared. Three additional metrics are defined to evaluate the properties of the value predictor themselves. The *Correct prediction rate* is the percentage of correct predictions out of the total number of instructions executed. Similarly, the *Misprediction rate* is the percentage of predictions that predict the wrong values out of the total number of instructions executed. *The Non-prediction rate* is the percentage of all instructions in which a prediction is attempted, but the predictor does not return a value. This type of nonprediction occurs when the confidence counter is below the predefined threshold required to make a prediction, or when the instruction being queried is not in the table, and so causes a table miss. The sum of these three rates reflects the bandwidth that a value predictor can provide. However, only the *Correct prediction rate* and the *Misprediction rate* influence the overall processor performance.

The correct predictions can improve performance by breaking true data dependencies. Mispredictions, on the other hand, may decrease performance since instructions that have been previously issued with an incorrectly predicted value must be squashed and reexecuted. Note that not all the correct predictions actually produce a performance improvement, though. As a result, there is not necessarily a proportional relationship between the speedup and the *Correct prediction rate.* The value of the misprediction penalty is set to one clock cycle to correspond to the misprediction penalties commonly assumed in other studies [3,11].

5. Analysis of Results

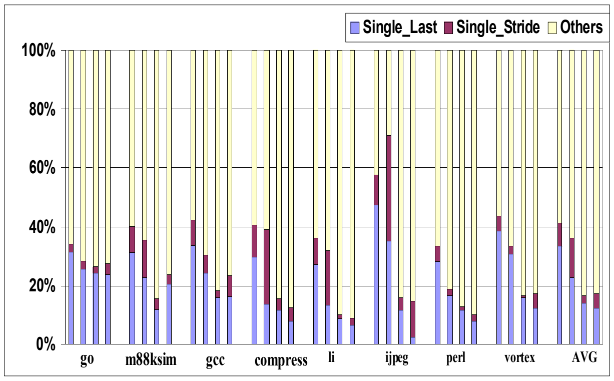
In this section, we first compare the classification capabilities of the real GCC compiler implementation to the idealized profile-based compiler. We then compare the overall processor performance obtained when using the different value predictors assuming no resource constraints. Finally, we compare the performance obtained with the different predictors when the number of read-write ports in each predictor is limited.

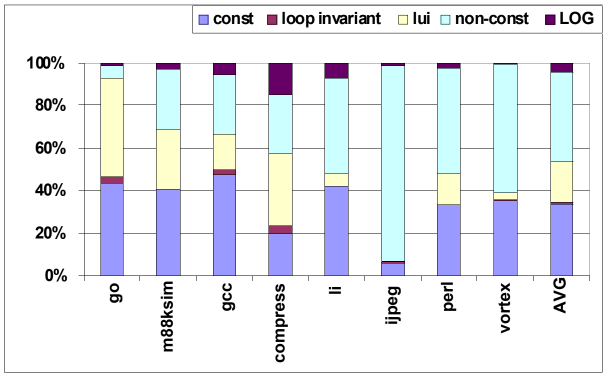
5.1 The capability of the real compiler

Figure 4a compares the classification capabilities of both the idealized compiler and the real compiler that uses only function-level information. The **Others** category in this figure represents all instructions that are not Single_Last or Single_Stride. It is observed that the real compiler is able to identify an average of approximately half of the instructions classified as Single_Last and Single_Stride by the idealized compiler. For *go*, the real compiler performs almost as well as the idealized compiler. For *ijpeg*, however, the real compiler can identify fewer than 30% of the instructions in the Single_Last and Single_Stride categories combined, even though these two categories count for approximately 50-70% of all of the instructions identified by the idealized compiler.

There are two primary parts of the Single_Last and Single_Stride categories that the real compiler cannot readily identified. The first part consists of those instructions that have invariant original definitions where the type of original definition is one of GLOBAL, ARG, RET or SP. (Recall that our algorithm identifies as Single_Last only those instructions whose types of all original definitions are

CONST.) The second part that the compiler cannot readily identify are those instructions that appear in function prologues or epilogues, denoted LOG. It is beyond the capability of a compiler with only function-level analysis to classify these types of instructions since this classification requires interprocedural information.





(a) (b)

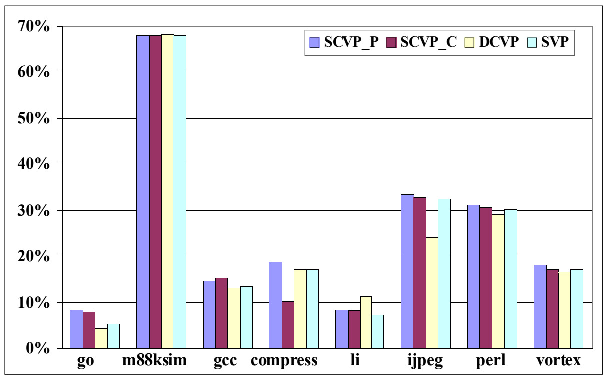
Figure 4. (a) The distribution of the value predictability pattern identified by the idealized compiler and the GCC-based compiler analysis. The four bars in each program show the following, starting at the left-most bar in each group: 1) static instruction count, idealized compiler; 2) dynamic instruction count, idealized compiler; 3) static instruction count, GCC-based compiler; 4) dynamic instruction count, GCC-based compiler.

(b) The dynamic instruction distribution of instructions in the Single_Last category using the GCC-based compiler showing the source of the Single_Last value predictability behavior.

Figure 4b shows a detailed distribution of the types of instructions in the Single_Last category as identified by the real compiler implementation. Each bar in this figure is divided into five components (from bottom to top): 1) instructions with only CONST original definitions; 2) those that are loop invariant computations within the current function; 3) those whose op-code is “lui”; 4) those with invariant original definitions other than CONST; and 5) those that appear in the function prologue or epilogue. From this figure, we can see that, on average, around 45% of the instructions in the Single_Last category have original definitions that are something other than CONST. The *ijpeg* program is the exception having this portion above 90%. On average, another 5% of the instructions in the Single_Last category appear in function prologues and epilogues. In the *go* program, fewer than 10% of the instructions in the Single_Last category have an invariant original definition that is other than CONST or that is in a function prologue or epilogue. As a result, most of the Single_Last instructions in this program can be identified by a function-level compiler.

5.2 Performance comparisons with unlimited resources

Figure 5a shows the speedups obtained with no resource constraints when using the four value predictors compared to the baseline processor with no value prediction. In particular, there are a sufficient number of read/write ports in each predictor in these simulations to ensure that there are no conflicts among instructions trying to access the predictor simultaneously. We can see that the performance of all three schemes is not substantially different in most cases. We do see that the performance of the DCVP scheme, which dynamically classifies instructions and distributes them to the various component predictors, exceeds the other schemes only for *li*, though, while it performs worse than the other schemes in five of the eight programs.



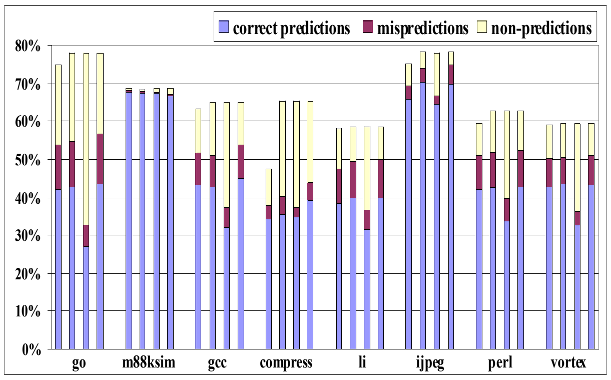


Figure 5. (a) The speedup over the baseline processor for the different value predictor mechanisms tested. (b) The corresponding value predictor characteristics. From left to right, the four bars in each program group are: SCVP_P, SCVP_C, DCVP and SVP.

In almost all cases, the idealized profile-based compiler classification scheme, SCVP_P, produces the best performance. The exception is that the real compiler-based scheme, SCVP_C, slightly exceeds the profile-based scheme for *gcc*. This anomaly occurs because in the SCVP_C scheme, all instructions that are not classified as Single_Last or Single_Stride are classified as **Others**. All instructions in this **Others** category then are predicted by the context-based predictor. In the SCVP_P, though, instructions categorized as Unknown are never predicted. Thus, the SCVP_C scheme actually attempts to predict more instructions than the SCVP_P scheme. In this particular program, these extra predictions produced slightly better performance for the SCVP_C scheme compared to the SCVP_P scheme. Except for *compress* and *gcc*, the performance of the real compiler scheme is almost the same as that of the profile-based compiler scheme. This shows that, in the absence of resource conflicts, the real compiler can do an excellent of classifying instructions into the component predictors, even when limited to function-level analysis.

Figure 5b shows that the corresponding correct prediction, misprediction, and nonprediction rates for the configurations compared in Figure 5a. We see from these results that, when there is no contention for read/write ports, the correct prediction rate of the DCVP mechanism tends to be somewhat lower than that of the other mechanisms. This lower correct prediction rate occurs because instructions are often dynamically reclassified by the DCVP, which also shows up in its higher nonprediction rate compared to the others. Consequently, the speedup obtained with the DCVP mechanism is lower than that obtained with the other mechanisms. It is interesting to note that the misprediction rate for the DCVP is the smallest among all the predictors. That is, the predictions produced by the DCVP tend to be more accurate than the other predictors, but it cannot predict as many instruction as the others due to the relatively large number of times it must reclassify instructions. This comparison suggests that higher overall performance can be obtained by predicting more instructions even with slightly lower overall accuracy.

5.3 Performance comparisons with limited read/write ports

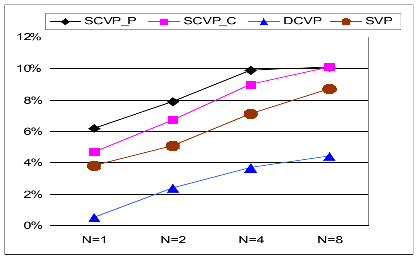
When the number of read/write ports in the value predictor is limited, some of the instructions that try to access the predictor will conflict with other instructions trying to access the predictor simultaneously. As a result, the prediction must be delayed to the next issue cycle, or the value will not be predicted at all. These conflicts will lower the correct prediction rate. Additionally, these conflicts will prevent retiring instructions from updating the predictor with the actual value produced in time for a subsequent prediction. This will introduce more stale values into the predictor which then will produce a higher misprediction rate. Both of these effects will decrease the performance produced when using a value predictor.

Figures 6 shows the speedup obtained compared to the baseline processor for the different value predictors when the number of read/write ports available, N, is varied from 1, 2, 4, to 8. From these figures, we can see that the profile-based compiler scheme, SCVP_P, provides the best speedup for all programs except *li*. The speedup produced when using the real compiler-based scheme, SCVP_C, is better than the DCVP scheme for half of the programs while it produces slightly lower performance than the DCVP scheme for the remaining programs. When N is 1 or 2, however, the SCVP_C scheme produces the same or slightly better performance than the DCVP scheme for six of the eight programs tested. The speedup for the SVP scheme is the lowest in almost all cases when N is limited. The major exception is that the DCVP scheme outperforms the others for the *li* program.

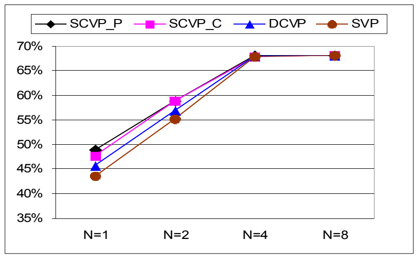
Recall that in the simple value predictor, SVP, each instruction to be predicted occupies a location in each of the component predictors. When the predictor is queried or updated, a read or a write must be made to each one of the three component predictors. In the compiler-directed static classification predictor and the dynamic classification predictor, on the other hand, each instruction occupies only a single entry in one of the component predictors. Thus, querying or updating these predictors requires only a single read or write operation to one of the component predictors, not to all three simultaneously. As a result, when the number of read/write ports is limited, the classification predictors will tend to produce higher prediction accuracy and fewer nonpredictions than the SVP. This effect is shown in Figure 7 as the number of read/write ports is varied.

It is clear from these results that both the static classification scheme and the dynamic classification scheme can provide more correct predictions than the SVP scheme when the number of read/write ports, N, is constant. This difference tends to increase as the number of read/write ports is decreased. The DCVP scheme tends to make more accesses to the predictor than the SCVP scheme, and it does not make effective use of these accesses, as shown by its higher nonprediction rate. This relatively inefficient use of the DCVP predictor resources occurs due to the frequent reclassification of instructions as they change their value predictability patterns during the program's execution.

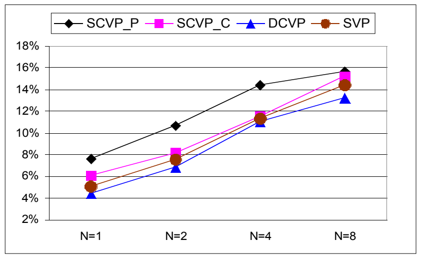
By distributing the instructions among its component predictors, the static classification scheme allows an average of about two times as many accesses to be made per cycle compared to the SVP scheme. Alternatively, for a given level of performance, the static classification scheme requires roughly one-half as many read/write ports in the predictor. Furthermore, the static schemes utilize the available read/write bandwidth more efficiently than the dynamic classification scheme. Consequently, in most of the cases, the profile-based compiler classification scheme, SCVP_P, has a higher correct prediction rate than the DCVP. The real compiler scheme, SCVP_C, produces a correct prediction rate very close to the DCVP scheme for a given number of read/write ports. Note that in the program *li*, though, the correct prediction rate of the DCVP is close to that of the SCVP_P scheme. However, it produces a lower misprediction rate than the SCVP_P scheme, which causes the DCVP scheme to provide the best speedup for this program.



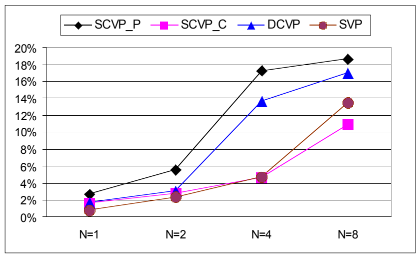
(a) go



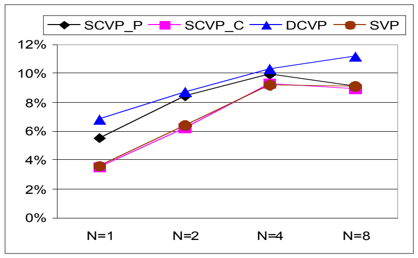
(b) m88ksim



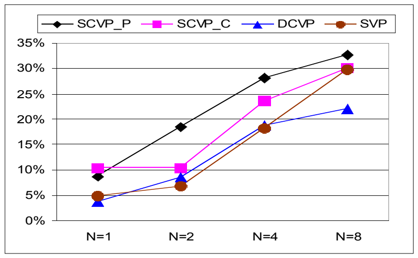
(c) gcc



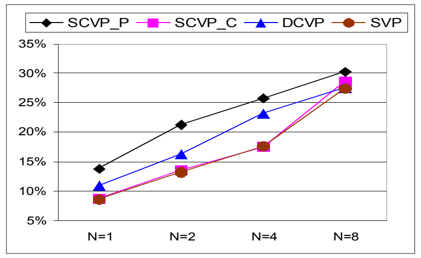
(d) compress



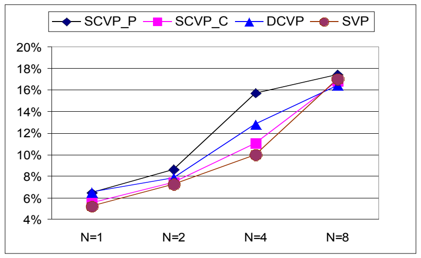
(e) li



(f) ijpeg

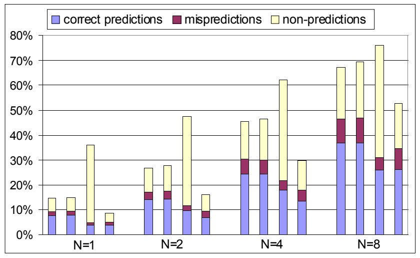


(g) perl

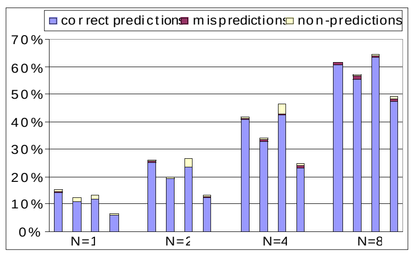


(h) vortex

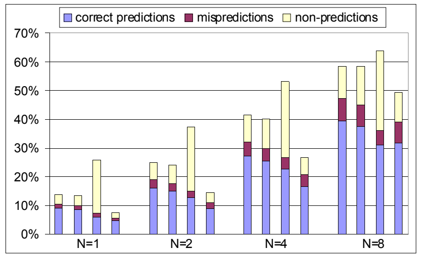
Figure 6. The speedup of different value predictors when N is varied



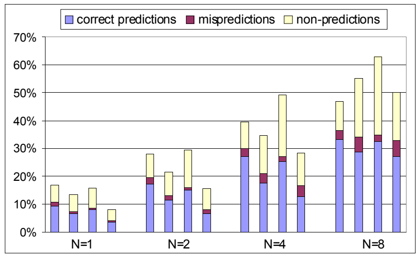
(a) go



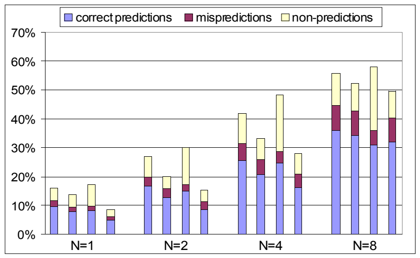
(b) m88ksim



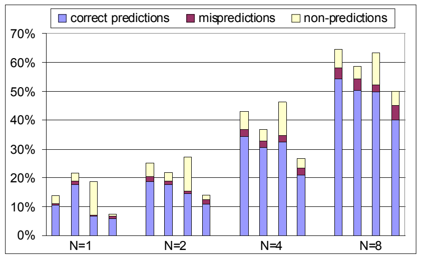
(c) gcc



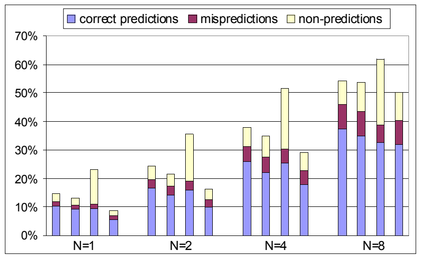
(d) compress



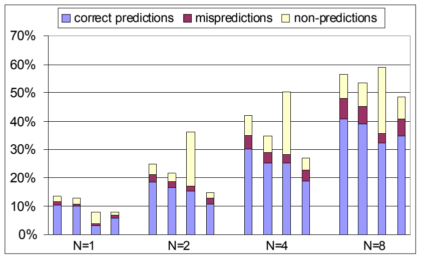
(e) li



(f) ijpeg



(g) perl



(h) vortex

Figure 7. The value predictor properties when N is varied. Four bars for each program denote SCVP_P, SCVP_C, DCVP and SVP from left.

Finally, it is clear that there is a nontrivial performance difference between the idealized compiler (SCVP_P) and the real compiler (SCVP_C) implementations when the number of read/write ports is limited. This difference suggests that more information is needed by the function-level compiler to match the potential performance shown by the idealized profile-based compiler. We believe that the use of interprocedural information will close this performance gap between the idealized compiler and the actual GCC compiler implementation.

6. Conclusion

This paper has studied the potential of a new compiler-directed classification scheme for statically identifying the value predictability patterns of the individual instructions in a program. Using a few extra bits in the instruction format, the compiler marks each instruction with the type of predictability pattern it is likely to exhibit when the program is executed. The algorithm for this marking is based on simple extensions to traditional compiler optimization algorithms for identifying loop invariant code and loop induction variables. The marked predictability patterns are used by the hardware to distribute the instructions at run-time to one of several different value predictors, including a last-value predictor, a stride value predictor, and a hybrid predictor. We implemented the classification algorithm both in an idealized profile-based compiler and in a version of the GCC compiler that has no interprocedural analysis capabilities. These two different implementations are used to demonstrate approximate bounds on how well a compiler can appropriately mark value predictability patterns.

We developed a simulator based on the SimpleScalar tool set to evaluate the performance potential of this static classification value prediction mechanism using several of the SPEC95 integer benchmark programs. The speedup obtained when this value predictor is incorporated in a 16-issue superscalar processor is compared to that obtained with a simple value predictor that predicts the values produced by an instruction using all three component predictors simultaneously, and with a value predictor that dynamically assigns individual instructions to the component predictors using only run-time information. Our simulation results indicate that, for a given level of performance, our compiler-directed static partitioning of instructions among the component value predictors can substantially reduce the number of read/write ports needed in the predictor. It is not unusual to see the number of ports needed reduced by a factor of two for many of the benchmark programs. Furthermore, for a given number of read/write ports, this static classification approach usually produces better overall processor performance than the dynamic classification approach.

This static classification approach thus reduces the cost and complexity of the value predictor. Several smaller value predictors can be built with only a limited number of read/write ports while still providing equivalent or better performance compared to a single value predictor with a larger number of ports. Furthermore, the classifications of predictability patterns and the corresponding compiler algorithms developed in this work more clearly show than any previous work the connection between the value locality behavior of an executing program and the source-level program structures that lead to this behavior. This connection leads to a deeper understanding of the causes of value locality and should suggest new directions for enhancing and exploiting this phenomenon.

Acknowledgements

This work was supported in part by National Science Foundation grants no. MIP-9610379, EIA-9971666, and CCR-9900605, by Kuck and Associates, Inc., and by the Minnesota Supercomputing Institute.

References

[1] D. Burger, T. Austin, and S. Bennett. The Simplescalar Tool Set, Version 2.0. Technical Report 1342, Computer Science Department, University of Wisconsin, Madison.

[2] F. Gabbay, and A. Mendelson. "The Effect of Instruction Fetch Bandwidth on Value Prediction." In Proceedings of the 25th Int'l Symposium on Computer Architecture(ISCA), 1998.

[3] S. Lee, Y. Wang, and P. Yew. "Decoupling Value Prediction on Trace Processors." In Proceedings of the 6th International Symposium on High performance Computer Architecture, 1999.

[4] S. Cho, P. Yew, and G. lee. "Decoupling Local Variable Accesses in a Wide-Issue Superscalar Processor." In Proceedings of the 26th Int'l Symposium on Computer Architecture (ISCA), Atlanta, May 1999.

[5] D. Connors and W. Hwu. "Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results." In Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO), Nov. 1999.

[6] M. Lipasti, and J. Shen. "Exceeding the Limit via Value Prediction." In Proceedings of the 29th International Symposium on Microarchitecture (MICRO), Dec. 1996.

[7] F. Gabbay, and A. Mendelson. "Speculative Execution Based on Value Prediction." EE Department Technical Report 1080, Technion-Israel Institute of Technology, Nov. 1996.

[8] Y. Sazeides, and J. Smith. "The Predictability of Data Values." In Proceedings of the 30th International Symposium on Microarchitecture (MICRO), Dec. 1997.

[9] K. Wang, M. Franklin. "Highly Accurate data value Predictions using Hybrid Predictor." In Proceedings of the 30th International Symposium on Microarchitecture (MICRO), Dec. 1997.

[10] B. Rychlik, J. Faistl, B. Krug, A. Kurland, J. Jung, Miroslav, N. Velev, and J. Shen. "Efficient and Accurate Value Prediction Using Dynamic Classification", Technical Report of Microarchitecture Research Team in Dept. of Electrical and Computer Engineering, Carnegie Mellon Univ., 1998.

[11] B. Rychlik, J. Faistl, B. Krug, and J. Shen. "Efficacy and Performance Impact of Value Prediction." Parallel Architectures and Compilation Techniques, Paris, Oct. 1998.

[12] B. Calder, G. Reinman, and D. Tullsen. "Selective Value Prediction." In Proceedings of the 26th International Symposium on Computer Architecture, May 1999.

[13] F. Gabbay and A. Mendelson. "Can Program Profiling Support Value Prediction?" In Proceedings of the 30th International Symposium on Microarchitecture (MICRO), Dec. 1997.

[14] Y. Sazeides, and J. Smith. "The Predictability of Data Values." In Proceedings of the 30th International Symposium on Microarchitecture (MICRO), Dec. 1997.

[15] A. Sodani, and G. Sohi. "An Empirical Analysis of Instruction Repetition." In Proceedings of 8th International Conference on Architectural Support for programming Languages and Operating Systems, October 1998.

[16] M. Johnson. Superscalar Microprocessor Design, Prentice Hall, 1991.

[17] S. S. Muchnick. Advanced Compiler Design Implementation, Morgan Kaufmann Publishers, Inc. San Francisco, California.