Speculative Execution based on Value Prediction

Freddy Gabbay

This work is also available as an internal EE Department Technical Report #1080, November, 1996.

Speculative Execution based on Value Prediction

Research Proposal towards the Degree of

Doctor of Sciences

Freddy Gabbay

Technion - Israel Institute of Technology, Electrical Engineering Department.

Contents

1. Introduction........7
2. The limits on instruction-level parallelism in serial code and current methods for parallelization........10
2.1 The limits on ILP........11
2.2 Current methodologies for exploiting ILP........14
2.2.1 Exploiting ILP within a basic block. 14
2.2.2 Exploiting ILP beyond a single basic block........18
2.2.3 The dataflow graph limits........22
3. Value prediction and its potential to exceed the dataflow graph limits........24
4. Various value predictor schemes........30
5. Related works........31
6. Simulation environment........36
7. Characterization of value predictability........37
7.1 Value prediction accuracy........37
7.2 Distribution of value prediction accuracy........43
7.3 Non-zero strides and immediate operations........45
8. Characteristics of value predictability in load instructions........49
9. The impact of value prediction on ILP........52
9.1 Various value prediction policies and the impact of the instruction window size........53
9.2 The impact of filtered value prediction........56
10. Conclusions and research plan........57
10.1 Contributions........58
10.2 Research plan........59
10.2.1 Further study on the characteristic patterns of value predictability........59
10.2.2 Combining program profiling and compiler support........59
10.2.3 Developing hybrid predictors and value generation modes........60
10.2.4 Microarchitecture aspects........61
10.2.5 Analytical model........61
10.2.6 Combining trace cache with value prediction for future processor architecture........61

Table of Figures

Figure 2.1 - A sample C program segment....11
Figure 2.2 - A control flow graph....15
Figure 2.3 - A dataflow graph of a basic block (dynamic dataflow graph)....17
Figure 2.4 - A dataflow graph of the entire program (static dataflow graph)....22
Figure 2.5 - The dataflow execution of the sample loop....24
Figure 4.1 - Last value predictor scheme....30
Figure 4.2 - Stride predictor scheme....31
Figure 4.3 - Register-file predictor scheme....31
Figure 4.4 - SEF predictor scheme....31
Figure 7.1 - Value prediction accuracy of integer load instructions in Spec-Int95....38
Figure 7.2 - Value prediction accuracy of ALU instructions in Spec-Int95....38
Figure 7.3 - Value prediction accuracy of integer load instructions in Spec-FP95....40
Figure 7.4 - Value prediction accuracy of ALU instructions in Spec-FP95....41
Figure 7.5 - Value prediction accuracy of floating point loads....42
Figure 7.6 - Value prediction accuracy of floating point computation instructions....42
Figure 7.7 - SEF prediction accuracy of floating point loads....43
Figure 7.8 - SEF prediction accuracy of floating point computations....43
Figure 7.9 - Predictability distribution of values....44
Figure 7.10 - 2-bit saturated counter for value prediction filtering....45
Figure 7.11 - The distribution of zero-strides out of overall instruction count. 47
Figure 7.12 - The contribution of Add/Sub immediate instructions to the successful predictions....47
Figure 7.13 - Addresses calculations out of successful predictions....48
Figure 8.1 - Address regularity versus value regularity out of the overall load misses....51
Figure 8.2 - Address regularity versus value regularity....51
Figure 8.3 - The comparable ILP of loads value prediction....52
Figure 9.1 - The ILP gained by value prediction when instruction window size is 40....54
Figure 9.2 - The ILP gained by value prediction when instruction window size is 200....55
Figure 9.3 - The effect of filtering on the gained ILP....56

List of Tables

Table 3.1 - The gained ILP when value prediction is employed (arrays were initialized with 0's).......................27

Table 3.2 - The gained ILP when value prediction is employed (arrays were initialized with random values).....28

Table 6.1 - The spec95 benchmarks..36

Table 7.1 - Summary of overall prediction accuracy..39

Table 7.2 - The distribution of non-zero strides...46

Abstract

This work presents a new approach for improving the parallelism that advanced computer architectures exploit at run-time from a sequential program (e.g. superscalar processors). The new approach termed *value prediction* suggests to predict outcome values of operations before they are executed and supply the predicted values to true-data dependent operations. As a result, the processor can speculatively execute the true-data dependent operations in parallel and extract instruction-level parallelism beyond the limits of the program's dataflow graph. The capability to use this new technique is based on new observations we made on the nature of computer programs. We revealed new patterns that can be exploited at run-time to predict of the outcome values of instructions.

The proposed mechanisms for value prediction are similar to the mechanisms being used for dynamic control speculation (branch prediction) and dynamic address speculation (prefetching). However they address other different issues that appear in modern microprocessors. Dynamic control speculation techniques aim at extracting parallelism from the sequential machine code beyond boundaries of a single basic block of instructions in order to gain as much parallelism as in the dataflow graph; while the goal of dynamic address speculation is to reduce effective access time by predicting which data items (addresses) would be needed by the processor.

This works presents a preliminary study of “value prediction”, broadly characterizes its related phenomena and the capability of using it to trigger speculative execution of truly dependent operations. Value prediction patterns are classified into two major classes: spatial value predictability and temporal value predictability. The effect of these patterns on the parallelism that can be exploited is presented as well. In addition various value predictors are introduced and their prediction accuracy for different data types is examined.

1. Introduction

Modern microprocessor architectures are increasingly designed to employ multiple execution units that are capable of executing several instructions, retrieved from a sequential program, in parallel. The efficiency of such architectures is highly dependent on the parallelism that their programs exhibit. Instructions within the sequential program cannot always be eligible for parallel execution due to several constraints. These constraints can significantly reduce the available parallelism which can be extracted from the program and, consequently, may limit the benefit of using such sophisticated architectures. We can classify these constraints into three classes: *true-data dependences*, *name dependences* and *control dependences*.

True-data dependences occur when an instruction generates a result that is required as an input by another instruction, and therefore these instructions can not be executed in parallel. Name dependences occur when instructions use the same register or memory location (*name*), but there is no flow of data between them as in true-data dependences. We can distinguish between two kinds of name dependences: *output dependences* and *anti-dependences*. Output dependence appears when instructions attempt to write simultaneously to the same name, while anti-dependence occurs when a *later* instruction attempts to write to a name before it is read by an *earlier* instruction. As long as name dependences are not handled, they can avoid instructions from being executed in parallel since the program correctness can be violated. Control dependences appear when the execution of instructions depends on the outcome of a branch (either conditional or unconditional). The outcome of a branch cannot be resolved instantly since it involves: 1. deciding whether the branch is taken or not (only in the case of conditional branch) and 2. computing its target address. As long as control dependences are not manipulated, control dependent instructions cannot even begin their execution because the branch outcome should determine the next sequence of instructions to be executed. Control dependences and name dependences are not considered an upper bound on the parallelism extractable from a sequential program, since they can be handled or even eliminated by various hardware and software techniques. Contrarily, only true-data dependences reflects the serial nature of a program by dictating in which sequence data should be passed between instructions. We shall formalize and broadly discuss these constraints in Section 2.

Basically, there are two fundamental approaches to executing multiple instructions in parallel: the superscalar processor and the VLIW (Very Long Instruction Word) processor. The superscalar approach extracts the parallelism from a sequential program at run time (*dynamically*), by employing sophisticated hardware mechanisms. Conversely, the VLIW approach tradeoffs this hardware complexity cost with advanced (*static*) compiler techniques. Both methods rely on the principle that true-data dependences form an upper bound on the potential parallelism that can be exploited from a sequential program. This kind of parallelism is represented by the *dataflow graph of the program*.

The superscalar approach provides a compatibility of the machine code between all the hardware implementations of a given instruction-set since it does not expose its internal parallel organization to the code generator. Its machine execution model is divided into two subsystems: instruction fetch and instruction execution. The two subsystems are separated by a buffer that is termed *instruction window*. The instruction fetch subsystem acts as the producer of instructions. It fetches multiple instructions from a sequential instruction stream, decodes them simultaneously and places them in program appearance order in the instruction window. The instruction execution subsystem acts as a consumer, since it attempts to execute instructions that have been placed in the instruction window. In several processors, the instruction execution

subsystem executes instructions according to their appearance order in the program (*in-order execution*). In this case it has to stall each time an instruction is not ready to be executed. As a result the processor can not look beyond the stalling instruction and detect ready-to-execute instructions. In order to overcome this limitation and better utilize its available resources, other superscalar architectures are capable of searching for candidate instructions beyond the stalling instruction and sending the ready instructions for execution. This kind of execution is termed *out-of-order execution*. When executing instruction out-of-order, the processor is compelled not to violate the correctness of the program, i.e. it should yield the same computation as if it had executed the program in a serial manner. The instruction window has a very important role in providing the superscalar processor the run-time look-ahead capability. As the size of the instruction window increases, it extends the look-ahead capability of the processor as well. Since the sequence of instruction execution in such superscalar processors can be faraway from an in-order execution sequence, they employ a special mechanism, termed *reorder buffer* ([John90]). The reorder buffer allows these processors to make the completion of instructions visible to the program, or the ”outside world”, in an in-order manner. Therefore, although instructions may complete their execution, they cannot completely retire since they are forced to wait in the reorder buffer until all previous instructions complete correctly. This capability is essential in order to keep the correct order of exceptions and interrupts that may occur during the execution, and to deal with control dependences as well.

The control flow of a program in a superscalar processor is achieved by a feedback between the instruction execution and the instruction fetch subsystems. This feedback determines the next instruction sequence to be fetched and decoded by the instruction fetch. As long as control dependences are not resolved, the instruction fetch subsystem cannot fetch instructions and provide candidate instructions to the execution subsystem. Various studies ([John90], [Rise72], [Nico84]) have shown that control dependences significantly reduce the parallelism that superscalar processors can extract from a sequential program. These works indicated that extracting parallelism only within a basic block (the code sequence between branch instructions boundaries) introduces limited performance gain, since basic blocks usually consist of a relatively small number of instructions exhibiting relatively poor parallelism. In order to better exploit parallelism, superscalar processors try to exceed control flow dependences. They employ a speculative execution of instructions, implying that the processor attempts to execute instructions that are dependent on an outcome of a branch before it is known whether these instructions should be executed. In order to speculate the next execution path, the processor employs *branch prediction* mechanisms ([Yeh91], [Yeh92], [Yeh93], [Smit81], [Smit84], [Fish92]). As a result, this technique can supply the execution subsystem with more candidate instructions for parallel execution by revealing parallelism across basic blocks. When the branch is completely executed (resolved), the processor should verify the correctness of the speculation. In case of an incorrect speculation, the processor should be capable of restoring the machine state. This capability is also provided by the reorder buffer since it guarantees that instructions which were executed speculatively will not retire and become visible to the program until their control path is validated. Several other hardware techniques have been developed in order to dynamically (at run-time) increase the parallelism that superscalar processors exploit. In order to reduce the impact of name dependences and even eliminate them, these processors employ special hardware mechanisms, invisible to the running program, allowing them to rename register identifiers at run-time ([John90]) and still maintain their correct semantics.

Unlike the superscalar approach, the VLIW approach has to rely on the exposure of its internal hardware organization (number of execution units, their execution latencies etc.) to the compiler. In this approach the compiler is responsible for the instruction scheduling (static scheduling) by packing independent instructions together as long-word instructions. This approach reduces, or even eliminates, the need for hardware scheduling mechanisms, and as a result significantly simplifies the hardware complexity and improves the cycle time of the processor at expanse of a more complex compiler. The outcome of the compiler scheduling in the VLIW depends on the hardware configuration of the processor. This information is important not only for the efficiency of the scheduling but also for the correct execution of the program. As a result, the instruction format of VLIW processors becomes implementation-dependent.

As well as the hardware simplicity of VLIW processors, their advanced compilers offer other advantages. When the compiler schedules instructions, it can examine and search for parallelism in a more extensive perspective compared to the hardware mechanisms of the superscalar processor. While superscalar processors maintain an instruction window with a restricted size, the compiler scheduler can evaluate the entire instructions in the program. In addition, in order to better exploit parallelism among instructions, the compiler scheduler of a VLIW processor can also schedule instructions speculatively across multiple basic blocks ([Elli86]) at compile time (sometimes it even requires a certain hardware support). The VLIW approach has also several disadvantages over the superscalar. Its compiler cannot depend on information that is available only at run-time. There are two typical examples illustrating this shortcoming: variable execution latencies and memory references disambiguation ([John91]). The first case occurs because the compiler cannot always know in advance the execution latencies of instructions, e.g., instructions that may be involved in cache miss or other unpredictable memory latencies. The second case occurs when instructions refer through pointers to memory storage locations but the compiler cannot decide whether these locations either completely or partially overlap. As a result of the inaccessibility to run-time information for the compiler, it is forced to apply conservative approaches when it schedules instructions. Due to this limitation the compiler cannot yield the best possible instruction scheduling. Unlike the VLIW, the instruction scheduling of the superscalar processor allows detection at run-time of such events as: cache misses, memory address disambiguation etc., and adapt the instruction scheduling accordingly. In addition, superscalar processors do not contradict the use of advanced compiler technology. In fact, several studies considered employing the advantage of the compiler scheduling for applications that are supposed to run on superscalar processors ([Chan91], [Hsu86], [Mahl92]).

This work presents a new concept termed *value prediction*, that is currently based on hardware support. Within this new concept, we claim that the bound of true-data dependences can be exceeded without violating sequential program correctness. This claim strikes two commonly recognized fundamental principles: 1. the upper bound of the parallelism in a sequential program is presented by the dataflow graph of the program, and 2. in order to guarantee the correctness of the program, true-data dependent instruction cannot be executed in parallel or in reversed order. The new approach attempts to collapse true-data dependences by predicting at run-time the outcome values of instructions before they are executed, and feeding the instructions that depend upon these values (true-data dependent) with the predicted values. As a result, value prediction may allow true-data dependent instructions to execute in parallel or even in reverse order.

From the hardware perspective value prediction is considered a feasible technique that fits the speculative nature of superscalar processors. It requires technology which is similar to the one used by branch prediction. Both value and branch

prediction require: (1) prediction schemes to generate the prediction, (2) scheduling mechanisms capable of tagging instructions executed upon a prediction, (3) a verification mechanism to validate the correctness of the prediction, and (4) a recovery mechanism that would allow the processor to recover from incorrect predictions. Even though both value prediction and branch prediction use similar mechanisms, we emphasize that the motivation for value prediction is substantially different from the motivation to perform branch prediction. While branch prediction aims at (1) increasing the number of candidate instructions for execution, since the amount of available parallelism within a basic block is limited [John90], and (2) helping to efficiently fill the pipeline bubble slots ([Henn91]); value prediction aims at allowing the processor to execute operations beyond the limit of true-data dependences (given by the dataflow graph).

This work initially reveals substantial evidence confirming that the computed values of a program are likely to be correctly predicted at run-time. In addition, it also studies various characteristics that are essential to allow efficient exploitation of the potential of this phenomenon, and finally it examines its impact on the parallelism that can be extracted. Since the concept of value prediction is entirely new to the society of computer architecture we focus our experiments on examining the capabilities of value prediction in an ideal execution environment, free from other limitations and constraints (such as the effect of control dependences, number of execution units) that can mask its potential. This approach is also strengthened by the work of Lipatsi *et al.* in CMU ([Lipa96a], [Lipa96b]) which indicates that a significant portion of the tremendous potential of this phenomenon is masked by existing processor limitations. However we are convinced that this phenomenon opens new horizons in microprocessor architecture research encouraging us to proceed in this direction.

The rest of the work is organized as follows: Section 2 discusses and formalizes the limits on parallelism and presents the current methodologies to extract parallelism. Section 3 introduces the concept of value prediction and shows how it can exceed current ultimate limits. Section 4 introduces several value predictors that we have developed. Section 5 discusses related works. Section 6 briefly presents the simulation environment we have used to perform our experiments. Section 7 broadly studies and analyzes the characteristics of value prediction. Section 8 presents characteristics of value prediction in load instructions. Section 9 presents the parallelism gain that value prediction can accomplish and Section 10 concludes this work and present the research plan.

2. The limits on instruction-level parallelism in serial code and current methods for parallelization

The parallelism that modern processors can extract from a sequential program is limited by both hardware and software constraints. In the previous section, we introduced the general principles and limitations of current approaches to exploit parallelism and our novel concept, value prediction. This section introduces formal definitions, in-depth discussion and various examples concerning the limitations on parallelism, and current methodologies to exploit them.

The measure of the parallelism that exists in a program is termed *instruction-level parallelism (ILP)* ([Henn90], [Henn95], [John90]):

Definition 1: ***Instruction-level parallelism (ILP)*** - a measure of the potential number of instructions that can be executed simultaneously in a program.

The ILP is constrained by several limitations caused by the relationships between instructions in a program. In order to evaluate the ILP we need to determine which instructions can be executed in parallel (simultaneously). A possible method of deciding this, is to examine whether a pair of instructions can be reordered without violating the correct execution of the program. If instructions can be reordered, they can also be executed in parallel, however, the opposite may not always be true (an example of such a case is presented in our further discussion). In addition, we consider a program to "execute correctly" only if its execution yields computation that is equivalent to the computation when it is executed it in a serial manner. A relationship between instructions that prevents them from executing in parallel is termed *dependence*. The instructions which are involved in a dependence between them are termed *dependent instructions*. Knowing the dependences between instructions in a program is important not only in order to evaluate the available ILP, but is also essential for the correct execution of the program.

In order to illustrate the following topics, we are assisted by a sample C program segment that is illustrated in figure 2.1. This code segment adds arrays B and C and stores the result in array A.

```
for(x=0;x<100000;x++) A[x]=B[x]+C[x];
```

Figure 2.1 - A sample C program segment.

Compiling the sample program with a C compiler which employs simple compiler optimizations[*], yields the following assembly code (for a Sun-Sparc machine):

```
(1)   22f0:  ld    [%i4+%g0],%l7      //Load B[i]
(2)   22f4:  ld    [%i5+%g0],%i0      //Load C[j]
(3)   22f8:  add   %i5,0x4,%i5        //Increment index j
(4)   22fc:  add   %l7,%i0,%l7        //A[k]=B[i] + C[j]
(5)   2300:  st    %l7,[%i3+%g0]      //Store A[k]
(6)   2304:  cmp   %i5,%i2            //Compare index j against threshold loop-termination value
(7)   2308:  add   %i4,0x4,%i4        //Increment index i
(8)   230c:  bcs   0xfffffff9  <22f0> //Branch
(9)   2310:  add   %i3,0x4,%i3        //Increment index k        (in branch delay slot)
```

2. 1. The fundamental limits on ILP

There are three types of elementary dependences between instructions that bound ILP: true-*data dependences*, *name dependences* (also termed *false dependences*) and *control dependences*. The type of true-data dependence is an inherent property of the program code. This kind of relation between instructions is defined as follows:

[*] The compilation was made with the '-O2' compiler optimization flag.

Definition 2: ***Instruction i is termed "true-data dependent on instruction j"*** - if either one of the following cases is met:

1. Instruction j produces a result that is consumed by instruction i.
2. There exists instruction k, such that instruction i is true-data dependent on instruction k, and instruction k is true-data dependent on instruction j.

An example of a true-data dependence of the first case type can be found in our sample program between instruction 4 (reads register %l7) and instruction 1 (writes to register %l7). In addition, we can also find an example of a true-data dependence of the second case type between instruction 5 and instruction 1. Instruction 5 (reads register %l7) is true-data dependent on instruction 4 (writes register $l7) and instruction 4 (reads register %l7 as well) is true-data dependent on instruction 1 (writes register %l7), therefore instruction 5 is also true-data dependent on instruction 1.

In current processor architectures, true-data dependent instructions cannot be executed simultaneously or completely overlapped since the processor cannot preserve the correctness of the program. Avoiding true-data dependences from violating the correct execution of the program can be accomplished by either hardware or software techniques. A possible hardware technique is to employ a pipeline interlock mechanism ([John90], [Henn90], [Henn95]) that stalls the dependent instructions until their input values are computed. A processor that does not employ such a hardware mechanism has to rely on the compiler to schedule the dependent instructions a sufficient distance from each other that would avoid the hazard. From the performance perspective, there are two different approaches to reduce the impact of true-data dependences:

1. Maintaining the dependence but avoiding the pipeline hazard. This approach is used by various techniques e.g.: dynamic code scheduling, static code scheduling, forwarding (bypassing) etc. ([Henn90], [Henn95], [John90], [Fish79], [Adam74]).
2. Eliminating the dependence by code transformation, e.g.: loop unrolling ([Weis87]).

Note that eliminating true-data dependences requires information about the global structure of the program and a considerable amount of analysis. Therefore it is maintained by the compiler for a very limited number of cases, in contrast to hazard avoidance which can be managed by both software and hardware.

We can conclude the importance of true-data dependence regarding the following issues:

1. It indicates the possibility of read-after-write (RAW) pipeline hazards ([John90], [Henn90], [Henn95]).
2. It determines the order in which results values must be calculated to yield a correct execution of the program.
3. We will show that true-data dependences set the upper bound on the amount of ILP that can be extracted by current architectures.

The second type of dependence, called name dependence, occurs when two instructions use the same register or memory location (termed *name*), but there is no flow of data between them. The relationship of name dependence between instructions is defined as follows:

Definition 3: ***Instruction i and instruction j are termed name dependent*** - if either one of the following conditions is met (assuming that instruction i precedes instruction j in the order of the program):

1. Instruction j writes to an operand name that instruction i reads. This type of name dependence is termed *anti-dependence*.
2. Instructions i and j write to the same name. This kind of name dependence is called *output dependence*.

Anti-dependence may cause write-after-read (WAR) pipeline hazard ([Henn90], [Henn95], [John90]). An examples of anti-dependence in our sample program appears between instruction 2 (reads register %i5) and instruction 3 (writes register %i5). Output dependence can cause write-after-write (WAW) pipeline hazard ([Henn90], [Henn95], [John90]). For instance, instruction 1 and instruction 4 in the sample program write to the same register (register %l5) and hence there exists output dependence between them (there is also a true-data dependence between them). These two kinds of dependences are termed "name dependences", as opposed to true-data dependences, since there is no data value that is transmitted between instructions. This means that name dependent instructions can be executed simultaneously, if the name used by the instructions is changed so that they do not conflict. Changing the name of register operands is termed *register-renaming* ([Henn90],[Henn95],[John90]). Register renaming can be done either statically by the compiler or dynamically by the hardware. Obviously, in a machine where the number of registers is unlimited, name dependences should not affect the ILP that the processor exploits. However in a more realistic machine, where the amount of registers is finite, it can cause pipeline stalls.

The third type of dependences is termed control dependence. Control dependences determine the ordering of instructions with respect to a branch instruction. The term "branch instruction" in our discussion refers to either conditional or unconditional branch (jump) instructions. The control dependences relations between instructions is defined as follows:

Definition 4: ***An instruction is termed control dependent on a branch*** - if it cannot be moved before the branch and still guarantee the correct execution of the program. An instruction is not control dependent on a branch if it *cannot* be moved after the branch ([Henn90], [Henn95]).

For instance, when referring to our code segment, all the instructions that appear in the program after the branch instruction (not illustrated in the code fragment) are control dependent on the branch instruction of the loop. Had this branch not been taken, they would not have been executed, and hence they could not be move before this branch. In addition, all the instructions that appear before the branch are not control dependent on that branch. If they had been moved after the branch, the correct execution of the program would not have been necessarily preserved. The later case is the classical example of independent instructions that cannot be reordered. Control dependences appear in programs because the processor can not know in advance what the outcome of branches will be. A branch outcome cannot be resolved instantly since it involves: 1. determining whether the branch is taken or not taken and 2. calculating the target address in case of a taken branch. As long as the branch outcome is unresolved the processor cannot decide what the next instructions sequence is to follow that branch. As a result it would have to stall from fetching any further instructions and executing them until the branch target is resolved. Preserving the control dependences is essential for the correct execution of the program, however this kind of dependences, unlike true-data dependence, is not an ultimate performance limit. Various hardware and software techniques

have been studied and employed during recent years to reduce the effects of control dependence, e.g. loop unrolling ([Henn90], [Henn95], [Weis87]), speculative execution ([Gwen95], [Diep95]) branch prediction ([Yeh91], [Yeh92], [Yeh93], [Smit81], [Smit84], [Fish92]), delayed branches ([Henn90], [Henn95]), predicated instructions ([Ando95], [Mahl95]). Some of these techniques are described in Subsection 2.2.

2. 2. Current methodologies for exploiting ILP.

Having described the fundamental limits of ILP, we turn to the second part of the discussion, where we illustrate in two phases how current processors exploit the available ILP and why they cannot exceed the dataflow graph limits. In particular, it will be shown that only ideal machines can accomplish these limits, while present processors can only seek to reach the true-data dependences limits.

2. 2. 1. Exploiting ILP within a basic block

A possible way to illustrate the control flow in a program is the *control flow graph* representation:

Definition 5: ***Control flow graph*** - is a directed graph that represents the possible paths of control flow in a program. Each node in the graph denotes a *basic block* and each arrow denotes a possible control flow path between basic blocks. The control path is determined according to the branch instruction at the end of each basic block.

Definition 6: ***Basic block*** - is a sequence of instructions with no branch-in except the entry and no branch-out except the exit. A basic block has a an important property - if any instruction in the basic block is executed, all other instructions are also executed.

The control flow graph representation and the partitioning of the code to basic block are essential for understanding the manner in which parallelism is extracted. Since there are no branches within a basic block (except an entry and exit), the dependences between instructions in it are known, except for the dependences between memory locations ([Henn90], [Henn95], [Elli86]). The control flow graph representation of our sample program is illustrated in figure 2.2.

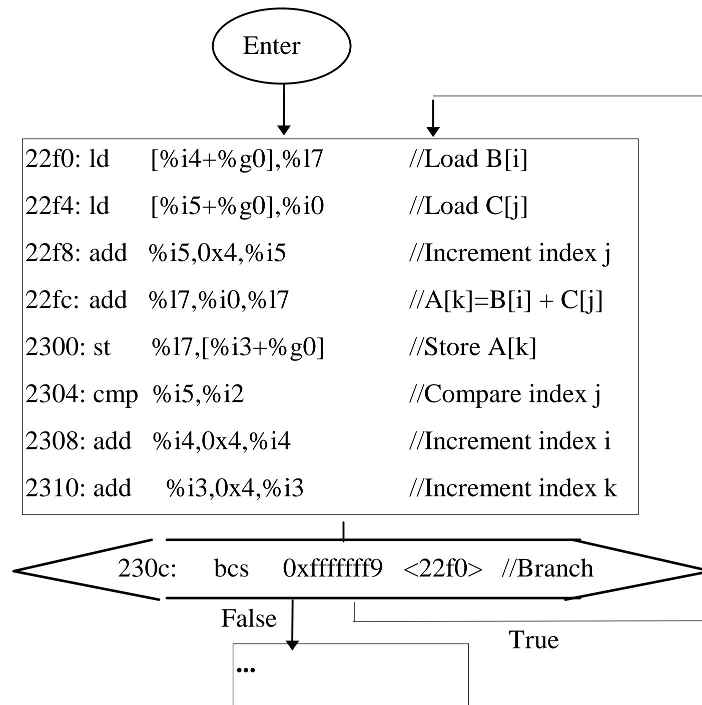

Figure 2.2 - A control flow graph.

Having partitioned the program into basic blocks, we shall start presenting the two major approaches to extract ILP: *dynamic scheduling* and *static scheduling*. The first is employed by the hardware at run-time, while the second is employed by the compiler at compile-time.

Dynamic scheduling is a method which is employed by the hardware in order to rearrange the order of instruction execution at run-time. This approach cannot remove true-data dependences. However, in order to eliminate some of their pipeline stalls and increase the utilization of the execution units, it allows instructions to execute *out-of-order* as far as the correctness of the program is preserved. If instructions were only allowed to be executed in the order in which they appear in the program's code (in-order execution), then whenever an instruction had a true-data dependence or name dependence, the processors would have to stall the execution of any further instruction. As a result the processor was unable to look ahead beyond the dependent instructions, even though there could have been subsequent instructions which were *ready-to-execute* and available execution units to accept them.

Definition 7: ***Ready-to-execute instruction*** - An instruction is considered ready to execute if all its source operands are ready (available).

To overcome the limitations of an in-order execution, the concept of out-of-order execution suggests isolating the pipeline execution stage from the rest of the pipeline stages, so that instructions are fetched from memory and decoded in order, attempted to be executed out-of-order, and *committed* in order.

Definition 8: ***Commit action*** - Committing an instruction means that the processor makes the completion of the instruction (modifying the registers-set, condition codes, exceptions and interrupts status etc.) visible to the program.

The isolation of the execution stage, allows instructions to be fetched from memory and decoded regardless of whether they can be executed instantly. In order to achieve this, a special buffer should be maintained between the decode stage and the execution stage. This special buffer is termed *instruction window* (also termed *reservation station*). Decoded instructions are sent to the instruction window as long as there is free space for them and they are placed there according to their program appearance order. All instructions that reside in the instruction window are examined as candidates for potential execution, and the ready-to-execute ones can be sent to their execution units. The look-ahead capability which is provided by the instruction window is limited by the size of the window, since it determines how far ahead the processor look-ahead can be. Several researches indicated that the effective size of the instruction window (the size that can be utilized efficiently by the processor) depends also on the capability to handle control dependences ([John90], [Rise72], [Nico84]).

Allowing instructions to be executed out-of-order implies that they may complete out-of-order as well, and as a result name dependences can violate the correct execution of the program, unless they are handled or eliminated. Dynamic scheduling can deal with name dependences through register renaming ([Kell75], [John90], [Henn90], [Henn95]). It dynamically maps the set of register identifiers known to the program to a set of physical hardware registers. Thus, the same register identifier appearing in several instructions can be mapped to different physical hardware registers. This kind of renaming allows the hardware to eliminate at run-time both output dependences and anti-dependences as long as it has free physical registers. For instance, the anti-dependence in our sample program between instruction 2 (reads register %i5) and instruction 3 (writes to register %i5), can be eliminated dynamically by mapping the virtual register identifier %i5 that appears in these instructions to different physical registers, say physical register 1 for instruction 2, and physical register 2 for instruction 3. This would allow both instructions to execute correctly out-of-order or even in parallel, however once these instructions are completed, the machine should maintain some book-keeping of the most recent value of the virtual register identifier %i5 residing in physical register 2.

Various techniques have been proposed during recent decades based on the concept of dynamic scheduling: The CDC6600 scoreboard ([John90], [Henn90], [Henn95]), the Tomasulo approach which also allows dynamic register renaming to avoid name dependences ([Toma67], [John90], [Henn90], [Henn95]), and various other similar mechanisms that appear in today's modern superscalar microprocessors ([Gwen95], [Diep95], [John90]). We can conclude that dynamic scheduling is an attractive approach because it offers the following advantages:

1. It can handle cases that can only be resolved at run-time.
2. It significantly simplifies the compiler.
3. It allows a code that was compiled without compiler optimizations to run efficiently on different pipeline organizations.

However, all these advantages are gained at a cost of hardware complexity of the processor.

Static scheduling can be a competitive alternative to dynamic scheduling, or even considered as an additional technique along with dynamic scheduling. Like dynamic scheduling, it aims at arranging the order of instructions in the program so that they can be executed correctly in the best order, or close to it. However, in contrast to dynamic scheduling where code rearrangement is made at run-time by the hardware, static scheduling is employed by the compiler. Eventually, static scheduling would increase the possibility that the processor would fetch independent instructions and would execute them more efficiently. One of the major advantages of this alternative is that it can considerably simplify the hardware

complexity of the processor in comparison to dynamic scheduling ([John90]). A basic static scheduling approach seeks to perform scheduling inside each basic block. In this case, the compiler partitions the program to basic blocks ([Aho86]) and constructs the *dataflow graph* of each basic block as illustrated by figure 2.3 for our sample program. A dataflow graph is defined as follows:

Definition 9: ***Dataflow graph*** - is a directed graph that represents the true-data dependences in a code sequence. Each node[*] in the graph indicates an individual atomic operation and each arrow denotes a true-date dependence between two nodes.

Currently, we restrict the dataflow graph to refer only to a sequence of instructions inside a basic block. Therefore a program can be referred to as a set of dataflow graphs each associated with a different basic block.

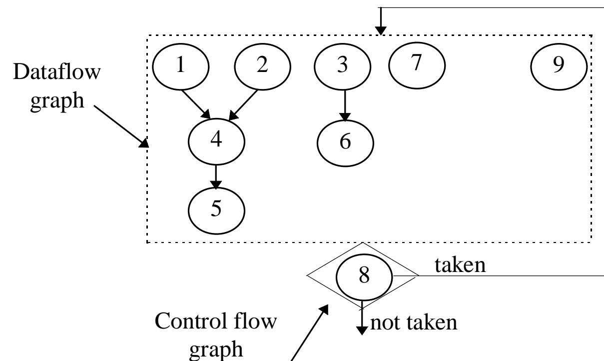

Figure 2.3 - A dataflow graph of a basic block.

The importance of dataflow graph representation is that it illustrates what instructions can be executed in parallel without violating the correctness of the program (currently we only refer to a basic block). The next step to be taken by the compiler is to use this information and examine the *critical paths* of each dataflow graph (of a basic block).

Definition 10: ***Critical path*** - is the path in the dataflow graph that has the longest execution time.

With respect to figure 2.3, if we assume that each instruction has the same execution time, instructions 1,4 and 5 form the critical path of that dataflow graph (also instructions 2, 4 and 5). The critical path identifies the sequence of instructions that is critical to the execution time of the code sequence (basic block). Once the dataflow graph and its critical paths are known, the compiler should try to ensure that the instructions in the critical path will be executed as fast as the length of the critical path allows and that the other instructions will be executed in parallel without disturbing them. When a machine has unlimited resources (in terms of execution units, memory ports etc.) the critical path is the path that dictates the execution time of the entire code sequence. However, when a realistic machine is considered (with a limited amount of resources) the scheduler needs to share the resources efficiently between the critical path and the other paths in the dataflow graph (of a basic block). Efficient sharing of the machine resources between the paths in order to ensure optimum scheduling may not be straightforward at all. Optimum scheduling can be gained by examining all possible schedules which satisfy the correctness of the program, however this was proven to be NP-Complete ([Coff76], [Moud94]), and hence it yields

[*] the node number denotes the index number assigned to each assembly instruction - see the output code generated by the compiler.

impractical compilation time ([Davi81]). Therefore most of the realistic algorithms of static code scheduling employ heuristic approaches such as *list scheduling* ([Adam74], [Blai94], [Fish79]), the "*first-come, first-served*" scheduler, the *greedy scheduler* and the *branch-and-bound scheduler* ([Davi81]).

Static code scheduling interacts with another important assignment of the compiler, *register allocation*. This kind of interaction may have a significant impact on the parallelism which the static scheduler can exploit. The aim of register allocation is to assign the machine registers to the program's variables in order to reduce the communication time with memory storage. It can be noticed that the goals of the static code scheduler and the register allocator conflict. On one hand the register allocator aims at allocating as few registers as possible to as many variables in order to decrease the possibility that there will not be enough available registers. On the other hand, the aim of the static code scheduler is to maintain as many independent instructions as possible to avoid name dependences, and as a result it increases the registers usage. This conflict is a fundamental design trade-off of the compiler. If the register allocation is performed before the code scheduling, the scheduler can be needlessly restricted by the register allocator. However if the static code scheduling and the register allocation are performed in the opposite order, the register allocator cannot guarantee that it can satisfy the amount of registers needed by the scheduler. This can result in a *spill-code* - a code fragment that transfers variables stored in registers to memory and vice versa. Spill-code can effect the benefits of efficient register allocation as well as the code scheduling severely. This effect as well as other issues concerned with the conflicts between the register allocator and the static code scheduler, and various algorithms to perform register allocation have been broadly studied in various works such as [Hwu88], [Chai81], [Good88] and [Chow90].

2. 2. 2. Exploiting ILP beyond a single basic block

It was observed in the previous subsection that either static or dynamic scheduling techniques (or even both) cannot exploit ILP beyond the limitations of the dataflow graph of a basic block (figure 2.3), since they could not move or schedule instructions across basic block boundaries. In addition, it is recognized that exploiting ILP within a single basic block boundary can be very limited, particularly where in many programs, basic blocks are too small to supply the processor enough independent candidate instructions to execute in parallel ([Henn90], [Henn95], [John90], [Rise72], [Nico84]). Due to these observations, control dependences become a crucial factor in the processor's performance equation. Therefore the next evolutionary step of many advanced processors was handling them in order to be capable of extracting parallelism across multiple basic blocks. Various hardware and software-based techniques have been proposed to reduce the effect of control dependences:

Hardware-based mechanisms attempt to dynamically (at run-time) predict the outcome of a branch before it is executed. This technique is termed *branch prediction* and the mechanisms that perform the prediction are termed *branch predictors*. Employing branch prediction allows the processor to guess what basic block will be executed next before the branch resolution. The processor can then proceed with fetching instructions from the predicted basic block and search for more candidate instructions to be executed in parallel. Consequently, the processor can execute several instructions belonging to different basic blocks simultaneously. This type of execution is termed *speculative execution based on branch prediction*, or shortly *speculative execution*.

Definition 11: ***Speculative execution (based on branch prediction)*** - An execution of a control dependent instruction before it is known whether it should be executed (before the branch they depend upon is resolved).

In order to guarantee the correct semantics of the program, speculative execution of instructions cannot be committed before the branch they depend upon is resolved. In case of an incorrect branch prediction, the processor should be capable of recovering from the misprediction and rolling back the machine state to the state prior to the branch, otherwise the correct execution of the program could not be guaranteed. This recovery procedure and refilling the pipeline with instructions from the correct branch target involves a certain penalty in term of processor clock cycles (this penalty is termed *branch misprediction penalty*). The branch misprediction penalty depends on various factors such as: the structure of the pipeline, the branch predictor type and the strategies that the processor employs in order to maintain the recovery. During recent years, various studies have examined many hardware schemes for branch prediction ([Chan96], [Diep95], [Ditz87], [Henn90], [Henn95], [John90], [McFa86], [Smit84], [Smit81], [Yeh91], [Yeh92], [Yeh93]), e.g. various branch-prediction-buffers schemes, the branch-target-buffer (BTB) which can employ either 1-level or 2-level prediction table, and the return-stack-buffer (RSB) which handles the prediction of indirect jumps and return-from-subroutine instructions.

We can conclude that the potential of speculative execution is highly dependent on the effectiveness of the two following factors:

1. The *branch prediction* accuracy that the predictor can accomplish

The branch prediction accuracy is defined as follows:

Definition 12: ***Branch prediction accuracy*** - is the number of successful branch predictions performed by the predictor out of the overall number of prediction attempts.

2. The cost when the branch is not predicted correctly (misprediction penalty).

Various software techniques performed by the compiler can also reduce the impact of control dependences. These methods can be divided into two categories: 1. methods that extract ILP across basic blocks that do not necessarily belong to loops and 2. methods to extract parallelism among different iterations of a loop:

The first method attempts to move and schedule instructions across basic block boundaries and still preserve the correctness of the program. Exceeding the boundaries of a basic block enhances the software scheduler with more flexibility to extract ILP, and provides a global perspective on the program. The quality of this kind of scheduling depends on the capability of the scheduler to evaluate critical paths spanning across multiple basic blocks. Such information can be very valuable to the scheduler since it can perform better scheduling decisions than knowing only the critical paths of each single basic block. This kind of scheduling across multiple basic blocks is termed *trace scheduling*. Trace scheduling combines two different phases:

1. *Trace selection* - attempts to detect a sequence of basic blocks that is likely to be executed by the program. This sequence of basic blocks is termed *trace*. The chosen trace will be optimized in the second phase.
2. *Global trace scheduling* - once a trace is chosen by the first phase, this phase attempts to compact the code within the selected trace into the shortest possible sequence of instructions that preserves the data and control dependences. This is

done by merging the basic block of the traces and performing basic block scheduling (list scheduling) on the new basic block.

The effectiveness of trace scheduling is also highly dependent on its ability to detect the likely traces to be executed by the program. “Gambling” on the right traces has a significant impact on the performance that trace scheduling can yield since the chosen traces are optimized at the expense of the other possible trace sequences. In order to predict what traces are likely to be executed, the compiler must employ some kind of prediction mechanism at compile time. Such prediction mechanism can be accomplished by either performing reasonable guesses (heuristics) or by using profiles of previous executions of the program. Once the trace is chosen, scheduling instructions across branches should be done very carefully in order to guarantee the correct execution of the program, particularly in respect to the following issues:

1. Control dependent instructions that are moved across branches can become speculative. If the trace was not predicted correctly by the compiler, the speculative instructions would not have been executed in the original program and as a result they may violate the correct execution of the program.
2. The scheduling phase should also preserve the correct data flow execution of the program. Moving instructions across basic block boundaries may change the data and name dependences of the original program.
3. Moving instructions across basic blocks should not change the original exception behavior and their appearance order in respect to the original program. Code movement may raise exceptions that would not have occurred in the original program or even change their correct appearance order.

The primary importance of trace scheduling is that it provides a scheme for reducing the impact of control dependences by moving the code across non-loop branches, while other compiler techniques that are presented in the following can only optimize the impact of loop branches. Various issues concerned to trace scheduling and its different approaches are broadly studied in various works such as [Blai94], [Elli86], [Fish79], [Fish81], [Fish83] and [Colw87].

The second software approach to deal with control dependences focuses on loops only. It attempts to extract ILP across different iterations of a loop. This purpose cannot be satisfied straightforwardly by the first technique since any attempt to move instructions from a basic block of a loop outside the loop and vice versa can violate the correct execution of the program. A possible method to evaluate the parallelism among iterations of a loop is to examine its *loop-level parallelism*:

Definition 13: ***Loop-level parallelism*** - is a measure to the parallelism that exists between different iterations of the loop. The loop-level parallelism is determined by the loop-carried dependences.

Definition 14: ***Loop-carried dependence*** - is a kind of true-data dependence between instructions in different loop iterations (sometimes the loop-carried dependence can also be a kind of name dependence but this case is not referred to in our discussion since previously described renaming techniques can resolve it.).

When our sample program (figure 2.1) is examined, it reveals from the viewpoint of the source code (C language) that all the iterations of the loop can be executed in parallel. In fact an equivalent code that performs the same computation could be illustrated as follows, where each statement (or iteration of the original program) can be executed in parallel:

```
A[0]=B[0]+C[0];
A[1]=B[1]+C[1];
A[2]=B[2]+C[2];
...
A[99999]=B[99999]+C[99999];
```

However, when we examine the assembly code of the program, it exhibits that different iterations of the loop cannot completely overlap because of loop-carried dependences. These loop-carried dependences are caused due to index incrementation, which is required for calculating the effective memory addresses of the load instructions and for evaluating the termination condition of the loop. In order to eliminate this kind of loop-carried dependence and exploit parallelism across multiple loop iterations, the compiler employs a loop transformation technique, termed *loop-unrolling*, that aims at transforming a loop body by replicating it multiple times and adjusting the termination code respectively. Hence, a single iteration of the unrolled loop is equivalent to multiple consecutive iterations of the old loop. Obviously, this kind of code transformation must preserve the correctness of the program, otherwise it cannot be employed. Loop-unrolling can be attractive because it offers the additional advantages:

1. It enlarges the size of the basic block of the loop and exposes instructions from different iterations to the software or hardware scheduler, so that it can schedule instructions more efficiently.
2. It reduces the branch instruction count, since multiple iterations of the original loop are now dominated by a single branch.
3. In many cases it can also reduce the number of index computation instructions, since they do not have to be replicated with the entire loop body. Instead, it can increment (or decrement) indexes only once at each iteration of the unrolled loop. This can be accomplished by scaling the stride value of the index incrementation to the number of iterations that were merged, and by adjusting the references to the index that appear in the addressing modes of load and store instructions correspondingly.

Loop-unrolling can also yield some disadvantages that can limit its benefits:

1. It increases the size of the program's code.
2. It increases the register usage, since the software scheduler needs to rename registers in order to eliminate name dependences across different iterations of the loop.
3. In several cases, the termination condition of the loop depends on data to be computed in the loop itself. This case can severely affect the software scheduler and make the unrolling quite complicated.

These aspects of loop-unrolling and other related issues are widely discussed in [Henn90], [Henn95], [John90] and [Weis87].

Software pipelining is another technique to exploit parallelism across multiple iterations of a loop. The goal of this technique is to execute loop iterations at the highest possible rate ([Henn90], [Henn95]). In order to reach this goal, it attempts to reorganize the loop such that each iteration in the new loop consists of instructions chosen from different iterations of the original loop. Unlike loop-unrolling, software pipelining does not enlarge the basic block of the loop since

a single iteration of the new loop does not contain replications of the original loop. Therefore the major advantage of this technique over loop-unrolling is that it consumes less code space and less registers. Both loop-unrolling and software pipelining improve the scheduling of loops, but they each reduces different kinds of loop overhead. Loop unrolling attempts to reduce the overhead of the branch and the index computations, while software pipelining attempts to reduce the *initiation time* to minimum.

Definition 15: ***Initiation time*** - is the time between the initiation (start) of a given iteration of a loop and the next iteration.

Broad studies on various software pipelining approaches are presented in [Char81], [Rau81], [Aike88], [Lam88] and [Lam89].

2. 2. 3. The dataflow graph limits.

In previous subsections we have presented the limitations on ILP and reviewed various software and hardware techniques to exploit ILP. We observed that one of the critical limitations on ILP is control dependences. We also indicated that these constrains are not an ultimate limit since their impact can be reduced by various techniques. However, even if these techniques could have eliminated all control dependences in the program, i.e. the control paths of the program would have been known to the processor in advance, we could not have gained infinite ILP. In this case the entire program turns to "a single basic block" since all branches are eliminated. Within this "new basic block" the parallel execution of instructions is still constrained by the true-data dependences between them. True-data dependences can be illustrated by a dataflow graph, this time referring to the entire program and not only to each single basic block separately. *This dataflow graph was recognized and considered as the supreme limit on ILP that can ever be gained by current processors*. In order to illustrate the dataflow graph of our sample program, we replicate our loop 100000 (according to the number of loop iterations) and ignore the effect of the conditional branch on the control flow. The dataflow graph of the entire sample program which is obtained is illustrated by figure 2.4.

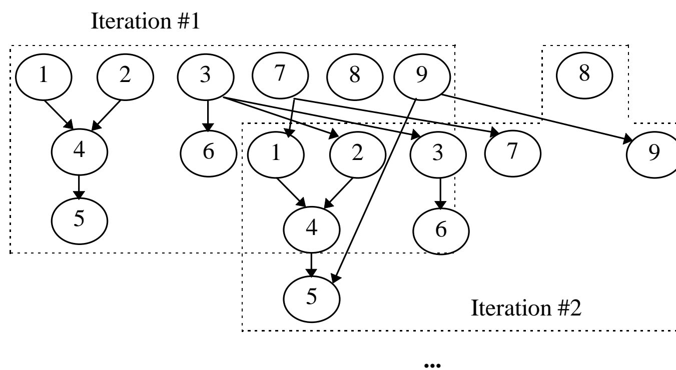

Figure 2.4 - A dataflow graph of the entire program (static dataflow graph).

In general, if a machine aims at exploiting ILP as such a dataflow graph exhibits, it should necessarily satisfy the following conditions:

1. It should know the control flow paths of the program in advance in order to eliminate control dependences.

2. Its resources in term of execution units, register file ports, memory ports etc., should satisfy the needs of the program in order to eliminate structural conflicts.
3. Its amount of register should be sufficient in order to satisfy all name dependences.
4. Its *instructions window* size should be big enough to evaluate all the instructions that appear in the dataflow graph.
5. It should be capable of fetching enough instructions from memory to satisfy the number of execution units, i.e. the bandwidth of the instructions fetch should be sufficient not to stall the machine from executing instructions. This capability is essential since the sequence in which instructions are stored in memory may not necessarily correspond to the execution sequence that is illustrated by the dataflow graph.

Generally, a machine that satisfies all these requirements is considered an *ideal machine*, since the only limitation that prevent it from extracting unlimited amount of parallelism is the flow of data among instructions in the program (true-data dependences). Such limitation is an inherent property of the program, as opposed to all other limitations that also depend on the machine architecture. While an ideal machine can reach the dataflow graph limitations, current realistic processor architectures can only seek to approach these dataflow graph boundaries, and therefore, they were also termed *restricted dataflow machines* ([Simo95]).

One may notice that this dataflow graph exhibits that different iterations of the loop do not completely overlap. The loop-carried dependences that prevents us from exploiting loop-level parallelism are caused by the index computations, which are required to compute the effective memory addresses of load instructions. This type of loop-carried dependence was referred to in our previous discussion, where it was mentioned that it may be resolved by employing loop-unrolling. However, even if loop-unrolling was considered it could not remove the true-data dependences inside the basic block of the loop. In addition loop-unrolling has a few disadvantages that were previously mentioned such as: 1. significantly enlarging the code size, 2. sometimes incapable of evaluating the number of loop iteration on compile time. Hence, in order to avoid arguing about the advantages and disadvantages of this technique we will leave the original code as it was generated by the compiler, but we will remark at this point that some software techniques can eliminate simple loop-carried dependences by their own limitations.

After constructing the dataflow graph of our sample program, we use figure 2.5 to calculate the ILP when the program runs on an ideal machine. This figure exhibits the overlapping between different iterations of the loop during the execution of the program. The instruction count of the entire program is 900000 (each iteration consists of 9 instructions). The illustration of figure 2.5 exhibits that the program can be executed in 100,002 clock cycles if we assume that the execution of each instruction takes a single clock cycle. Hence the ILP presented by the dataflow graph of the entire program is 900000/100002= 8.99 instructions/clock cycle.

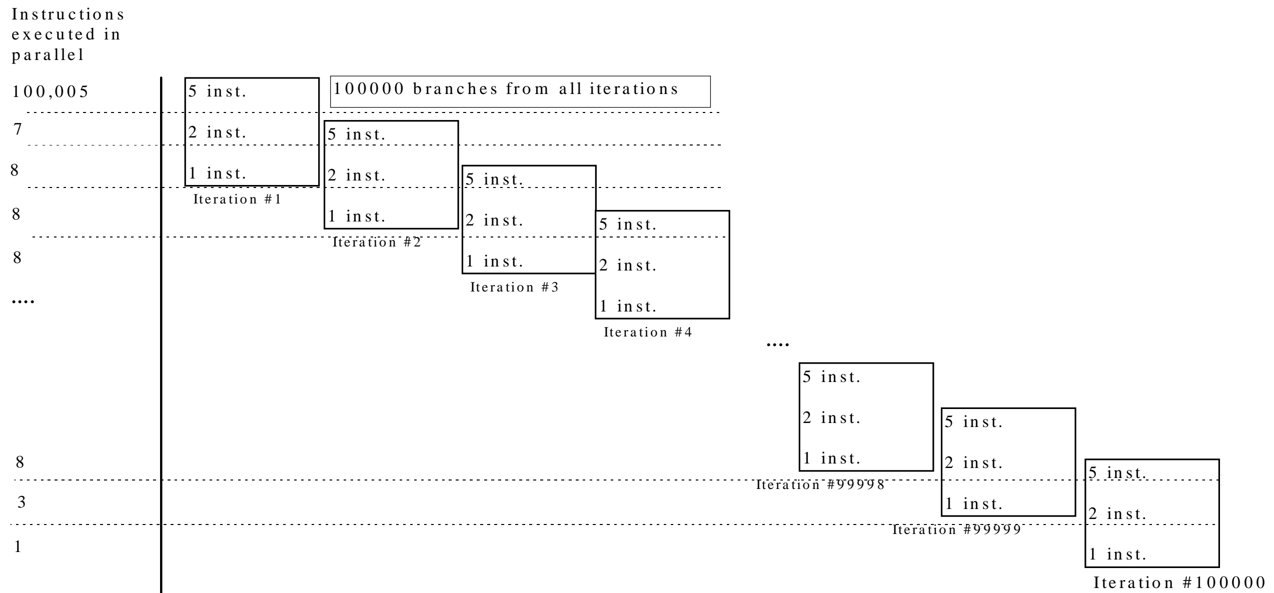

Figure 2.5 - The dataflow execution of the sample loop.

We summarize the discussion in this section with the following conclusions:

- The instruction-level parallelism in a program is limited by several constraints: true-data dependences, name dependences and control dependences.
- Some of these constraints can be efficiently handled by both software and hardware techniques. Some of these techniques can only extract ILP within a basic block while others can exceed basic block boundaries and take advantage of ILP across different basic blocks or loop iterations.
- Even if current processors were ideal, they could not eliminate true-data dependences and exceed the dataflow graph limitations.
- The true-data dependences in a program and its dataflow graph representation are considered as a fundamental ultimate bound on the instruction-level parallelism that current processors can exploit.

3. Value prediction and its potential to exceed the dataflow graph limits

Our previous discussions have broadly presented the current limits of ILP and various methods to extract parallelism. It was shown that even if current processors were ideal machines the ultimate strict limit on ILP was the dataflow graph of the entire program. This section describes the main concept of this work, termed *value prediction*. It will illustrate how this concept can exceed the fundamental ILP limits facing present processors - the dataflow graph. Before we define the notion "value prediction", we classify the instruction-set of the machine into two groups: the first group consists of *value generating instructions* and the second consists of all the rest. A value generating instruction is defined as follows:

Definition 16: ***A value generating instruction*** - is any kind of a machine instruction that produces a data value, such as assigning a result value to a register, memory item, condition codes, or any other storage item.

E.g.: load instructions, arithmetic instructions etc. are all value generating instructions since they assign value to a destination operand. We can now define the notion "value prediction" as follows:

Definition 17: ***Value Prediction*** - is a methodology to predict run-time outcome values of *value generating instructions* before they are executed. In general, the outcome value of an instruction is not restricted, thus it can be assigned to registers, memory locations, set of registers, condition codes etc.

The goal of value prediction is to exceed the limits of true-data dependences. It simply attempts to predict the outcome values of instructions before they are executed and feed the instructions that depend upon these values with the "guessed values". As a result, it may allow true-data dependent instructions to execute in parallel or even in a reversed order. In order to predict values that will be produced by instructions before they are executed this technique needs to employ a special hardware mechanism, a *value predictor*.

Definition 18: ***Value predictor*** - is a hardware-based mechanism that produces a predicted outcome value.

Various value prediction schemes can be employed to perform value prediction. A broad discussion of the various schemes that were developed in this and other work will be introduced in Section 4. For the current section we will avoid going into a detailed description of the internal specification of these mechanisms.

We term the execution of a true-data dependent instruction whose input values are supplied based on value prediction as *speculative execution based on value prediction* in contrast to *speculative execution based on branch prediction*.

Definition 19: ***Speculative execution based on value prediction*** - is an execution of a true-data dependent instruction that: 1. not all its input values have been computed yet and 2. all the unavailable input values are supplied by the value predictor.

Unlike the case of speculative execution based on branch prediction, where the execution of instructions is speculative because it is control dependent, in this case the execution is speculative since the instruction is true-data dependent and its dependence is not resolved until its execution. Both kinds of speculative execution aim to extract more parallelism in a program, and there is no contradiction to employing both of them. However the major difference between them is that speculative execution based on branch prediction seeks to schedule instructions in the manner they are presented by the program's dataflow graph by attempting to eliminate control dependences, while speculative execution based on value prediction attempts to exceed the dataflow graph limits. In addition, speculative execution based on value prediction implies that the execution sequence of instructions that was previously dictated by the dataflow graph, can be violated as long as the predicted values are verified in any manner that guarantees the correct execution of the program. As a result, machines that employ value prediction can violate the dataflow limits and 1. may gain better ILP than the dataflow graph exhibits and 2. still maintain the correct execution of the program. So far, when previous works on ILP have used the notion *speculative execution*, it was obvious to everyone that they all referred to the speculation on control flow, since the concept of value prediction had not been introduced then. Therefore from now on, each time this notion is used, we have to specify to which kind of speculative execution it refers.

From the hardware perspective, the concept of value prediction can be very practicable since it requires a very similar technology to that used by branch prediction mechanisms. A speculative execution based on either value or branch prediction requires prediction schemes to generate the prediction, scheduling mechanisms capable of taking advantage of

the prediction and tagging instructions that were executed speculatively, a validation mechanism to verify the correctness of the prediction and a recovery mechanism to allow the machine to recover from incorrect predictions. The availability of some of these mechanisms in current superscalar processors makes value prediction fit the speculative nature of these processors.

In order to illustrate how value prediction can work, we refer to the dataflow graph of our sample program that was illustrated in figure 2.4 and we perform various experiments to exhibit how it can be exceeded. These experiments measure the ILP of the sample program for the following cases:

- No value prediction is allowed.
- Value prediction is allowed only for loads.
- Value prediction is allowed only for ALU (arithmetic logic) instructions.
- Value prediction is allowed for both ALU and load instructions.

In addition, the ILP that is measured in each experiment is examined in two different size of instruction windows, 40 and 200 in respect to the different cases above. These simulation results are presented in tables 3.1 and 3.2. The first set of results refers to the case where all the data in the different arrays were initialized to the same value, say arrays of zeroes, before the execution, so that all the loaded values from memory can be predicted correctly on run-time (yet it does not matter how). The second table refers to the situation where the data in arrays was initialized (before the execution of the code segment) with random values, so that no specific pattern can be observed for loaded values. At this point it is important to clarify that such trivial cases, where all the components in the arrays have been initialized to the same value, can be handled efficiently by the compiler, if this value is known at compile time, otherwise the compiler cannot evaluate the loaded values. Moreover, even the simple loop-carried dependence that is caused due to the indexes calculation can be avoided by the compiler by employing loop-unrolling. However, the main purpose of presenting this trivial code example is not to show whether value prediction can or cannot beat other techniques to exploit ILP, but to provide a simple illustration how it can work.

The experimental results indicate that when value prediction is not used, both instruction windows gain the expected ILP that is equal to approximately 9[*]. This result is also equal to the ILP that was calculated out of the dataflow graph in Subsection 2.2.3. The enlargement of the instruction window size does not improve the ILP, since loop-level parallelism cannot be exploited due to the loop-carried dependences. When allowing value prediction for both ALU and load instructions, it resolves all true-data dependences since the indexes calculations can be always predicted correctly, as well as the add operations that add different components from the arrays and the load instructions. Therefore the ILP in this set of experiments (table 3.1) is only limited by the instruction window size, nearly 40 and 200 respectively. This case serves, as well, as a good example to illustrate how value prediction converts sequential code into a "vector like code". Additional results included in the first set of experiments indicate that value prediction of ALU instructions is more significant for the sample program, than value prediction of load instructions. When value prediction is allowed only for loads, no boost in the

[*] Also notice that the measured ILP may be slightly affected by initialization code and termination code not shown in the sample code.

ILP is observed. This observation is indeed accurate since the true-data dependences that are associated with the load instructions do not allow us to exploit loop-level parallelism across multiple iterations. The pair of load instructions in every basic block (iteration) limit the available parallelism within the basic block itself, however as long as the loop-level parallelism cannot be exploited, no further ILP can be gained. When value prediction is allowed only for ALU instructions it can still gain a significant boost in the ILP in respect to the case when value prediction is allowed for both load and ALU instructions. The explanation to this observation is that loop-level parallelism can be exploited due to the value prediction of the indexes that are computed by the ALU instructions. Only when loop-level parallelism is allowed to be exploited, value prediction of loads provides additional contribution to boost ILP as is illustrated in table 3.1.

	Instruction window = 40	**Instruction window = 200**
No value prediction	ILP=8.99	ILP=8.99
Load value prediction	ILP=8.99	ILP=8.99
ALU value prediction	ILP=35.99	ILP=179.9
Load and ALU value prediction	ILP=39.99	ILP=199.9

Table 3.1 - The gained ILP when value prediction is employed (arrays were initialized with 0's).

In order to further investigate the impact of the load instructions on the overall performance of this simple example, we repeat the experiments, but initialize the data in the arrays to random values, so that it becomes impossible to predict the outcome values of the load instructions. Notice that initializing the array with unpredictable values affects not only the outcome values of the load instructions that cannot be predicted correctly anymore, but also the outcome value of the add instruction, which adds the arrays components. In the previous case where the arrays were initialized with zero values, even if value prediction of load instructions was not allowed, the add instruction always generated zero values and hence it could be predicted correctly. However, in this case the results of the add instruction can no longer be predicted correctly, since it adds two random values. Therefore this case eliminates the capability to predict the load and the add instructions as well. The results of this set of experiments are summarized in table 3.2. We indeed observe that attempting to predict only the values of load instructions is clearly useless, since the value predictor cannot predict them correctly. When value prediction is allowed only for ALU instructions the ILP becomes nearly 30 when the instruction window size is 40, and 150 when the size is 200. The significant increase in ILP is again obtained due to the loop-level parallelism that can be exploited now. Employing both load and ALU value prediction does not gain ILP beyond the ILP gained by ALU value prediction since in both cases, neither the loads nor the add instructions of the array components can be predicted correctly.

	Instruction window=40	Instruction window=200
No value prediction	ILP=8.99	ILP=8.99
Load value prediction	ILP=8.99	ILP=8.99
ALU value prediction	ILP=29.99	ILP=149.9
Load and ALU value prediction	ILP=29.99	ILP=149.9

Table 3.2 - The gained ILP when value prediction is employed (arrays were initialized with random values).

The potential of the value prediction concept significantly depends on the *value prediction accuracy* that it can accomplish.

Definition 20: ***Value prediction accuracy*** - is the number of successful value predictions out of the overall number of prediction attempts gained by the value predictor.

Two different elements determine the value prediction accuracy: (1) the value predictor scheme and its capabilities (resources etc.), and (2) the nature of *value predictability* that resides within the program code.

Definition 21: ***Value predictability*** - is the potential that resides in a program to successfully predict the outcome values that it generates during its execution (out of the entire values that the program generates). This parameter is not restrained to a particular value predictor. It is an inherent property of the program itself, its databases and its computation algorithm.

The value predictor scheme determines the prediction formula, or the manner in which the prediction is made. Broad discussion of various value predictor schemes will be introduced in Section 4. In this discussion we focus on the second element which is an inherent property of the program and its data. We distinguish between two different varieties of value predictability: *temporal value predictability* and *spatial value predictability*:

Definition 22: ***Temporal value predictability*** - is a measure of the likelihood that an instruction will generate an outcome value as a function of the most recent *value* that it had generated.
Definition 23: ***Spatial value predictability*** - is a measure of the likelihood that an instruction will generate an outcome value as an extrapolation of its previously generated *values*.

E.g., when the arrays in our sample program were initialized with zero's, the load instructions in the program exhibited temporal value predictability since they always loaded the same value. The add instructions that incremented the indexes have always exhibited spatial value predictability since their outcome values could be predicted correctly based on the two most recently generated values. Substantial evidence to the existence of these distinctive phenomena will be presented in Section 7. In addition, we will show that categorizing value predictability to these two characteristics, temporal value predictability and spatial value predictability, is very significant, since computer programs may exhibit combinations of these properties.

Value prediction is not only a technique to boost instruction-level parallelism. It should also be regarded as an approach that reflects a distinctive phenomenon existing in computer programs - value predictability. Categorizing value

predictability to temporal and spatial properties recalls another different phenomenon, termed *memory locality* ([Henn90], [Henn95]). The concept of memory locality claims that most computer programs do not accesses memory items uniformly, since programs tend to reuse their data and code items. The memory locality that programs exhibit can also be categorized as two different properties: *temporal memory locality* and *spatial memory locality*. Temporal memory locality claims that recently used memory items are likely to be used in the near future, while spatial memory locality claims that memory items which are stored near each other (in terms of address location) are likely to be accessed close together in time. The classification of memory locality to temporal and spatial properties should not be confused with the classification of value predictability despite the resemblance between them. Unlike memory locality that is concerned with the reuse of the *location* of the storage items (memory addresses), value predictability refers to the reuse of their *content values* (the content values of registers and memory). In addition the connotation of the "spatial" and the "temporal" properties of value predictability refers to the space of values that the program generates, while the connotation of this terminology in memory locality refers to the space of the addresses accessed by the program. Value predictability and memory locality are also exploited for different goals. While value predictability is mainly taken advantage of in order to enlarge the instruction-level parallelism in a program, memory locality was widely exploited in order to improve memory access time by employing special *address-prediction* mechanisms (*prefetching*) such as [Pint96] and [Chen95].

Definition 24: ***Address Prediction (Prefetching)*** - is a technique to predict memory addresses generated by memory reference instructions (for instance load or store instructions) before they are executed. Its purpose is to fetch these items from memory before they are demanded in order to reduce the effective memory access time.

We make an extensive comparison and discuss the relations between value predictability and memory locality in Section 8. Before we turn to the next section to present various value predictor schemes, we close the discussion of this section with the following conclusions:

- We have introduced a new technique, termed value prediction, and illustrated its potential to: 1. exceed the dataflow graph limit and 2. preserve the correct semantics of the program without having to execute instructions in the sequence that the dataflow graph dictates.
- We have indicated that this technique can be feasible since it uses available technology.
- We have introduced a new terminology related to value prediction and extended some of the notions in the prior terminology:
    1. We distinguished between two different kinds of speculative execution: speculative execution based on branch prediction and speculative execution based on value prediction.
    2. We introduced the notion of value predictability and distinguished between two different types of value predictability temporal value predictability and spatial value predictability.

4. Various value predictor schemes

In this section four different hardware-based value predictor schemes are introduced: *last-value* predictor, *stride* predictor, *register-file* predictor and SEF predictor. The common role of these mechanisms is that they aim at predicting the destination values of instructions. In addition, all these predictors perform a dynamic and adaptive prediction, since they all collect and study history information at run-time, and with this information they determine their value prediction. The four schemes differ from each other in their prediction formula. The prediction formula determines the predicted value, i.e. the manner in which a predicted destination value is determined. It is important to clarify that at this point of the research, the main goal is not to discuss yet any particular hardware implementation of the predictor schemes. The purpose is to examine the characteristic of the value prediction phenomenon from a general viewpoint, i.e. the predictor schemes are discussed at a functional level without referring yet to detailed hardware implementation. In addition, for the sake of generality, even the size of the prediction table employed by these schemes is assumed to be unlimited in our experiments. Moreover, for the sake of simplicity, this discussion will only focus on value prediction where destination values are assigned to registers, even though all these schemes can be applied to memory storage location and condition codes as well.

Last-value predictor predicts the destination value of an individual instruction based on the last previously-seen value it has generated. In our simulation experiments this predictor is organized as a table (e.g. cache table - see figure 4.1), and every entry is uniquely associated with an individual instruction. Each entry contains two fields: *tag* and *last-value*. The tag field holds the address of the instruction or part of it (high-order bits in case of an associative cache table), and the last-value field holds the previously-seen destination value of the corresponding instruction. In order to obtain the predicted destination value of a given instruction, the table is searched by the absolute address of the instruction. Performing the table look-up in such a manner can be very efficient, since it can be done in the very early stages of the pipeline (the instruction address is usually known at fetch stage). A version of the last-value predictor (using an indexed but not tagged table) was also studied by Lipasti *et al.* in [Lipa96b].

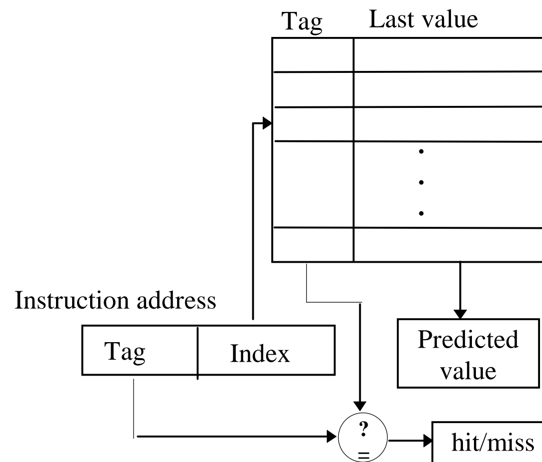

Figure 4.1 - Last value predictor scheme.

Stride predictor: predicts the destination value of an individual instruction based on its last previously-seen value and a calculated stride. The predicted value is the sum of the last value and the stride. Each entry in this predictor holds an

additional field, termed *stride* field that stores the previously-seen stride of the individual instruction (figure 4.2). The stride field value is determined upon the subtraction of two recent consecutive destination values.

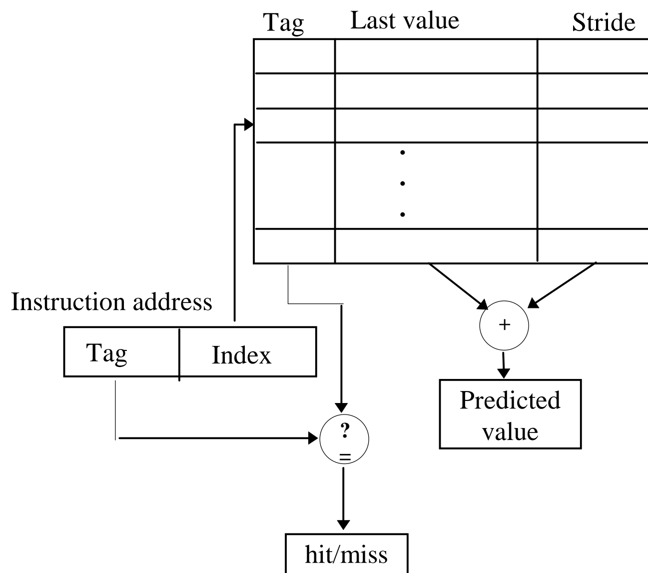

Figure 4.2 - Stride predictor scheme.

Register-file predictor: predicts the destination value of a given instruction according to the last previously-seen value and stride of its destination register (the recent value and the stride could have possibly been determined by different instructions). The register-file predictor is organized as a table as well, where each entry is associated with a different (architectural) register. The past information of each register is collected in two fields: a *last-value* field and a *stride* field. The last-value field is determined according to the last-value written to the corresponding register, and the stride value is determined according to the difference between two recent consecutive values that were written to the specific register (possibly by different instructions).

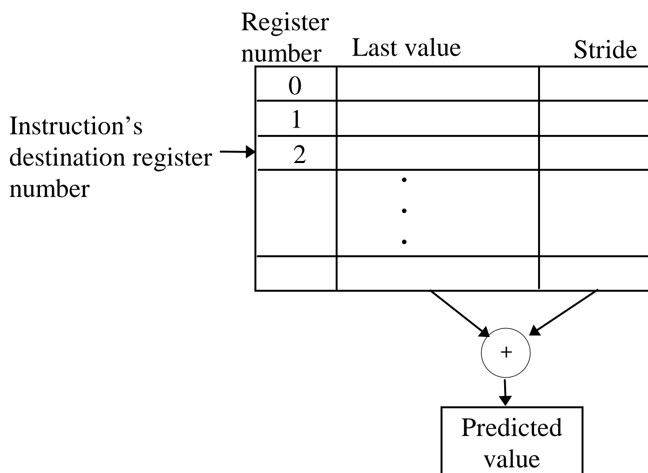

Figure 4.3 - Register-file predictor scheme.

It can be noticed that the last-value predictor can only take advantage of temporal value predictability since the prediction is made upon the last value, while the stride predictor can exploit both temporal and spatial value predictability that reside in the program. The register file predictor can be very attractive since its prediction table is relatively small. However since the predicted values are determined according to the history information of the register, there can be aliasing between different instructions that write to the same register. As a result, it may have serious influence on the prediction accuracy that it can accomplish.

SEF predictor: is used as an alternative predictor scheme for floating point values (instead of the three previous predictor). Floating point values are determined according to three different fields: sign, exponent and fraction ([Henn90], [Henn95]). Unlike the approach of the previous predictors, this predictor scheme attempts to predict the sign, exponent and fraction of floating point values separately. The predicted sign value is determined according to the previously-seen sign value, the predicted exponent value is determined according to the previously-seen exponent value plus a previously-seen exponent stride and the predicted fraction value is determined similarly (previously-seen fraction plus previously-seen fraction stride). The motivation to perform a prediction for floating-point data types in such a manner is discussed in Section 7.

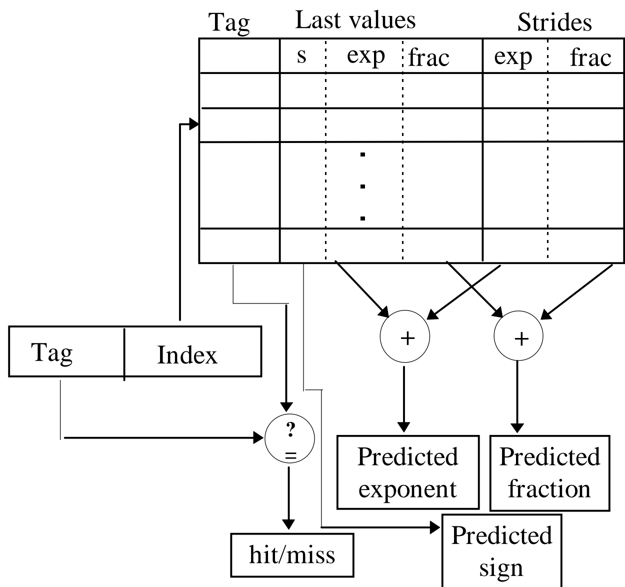

Figure 4.4 - SEF predictor scheme.

5. Related works

Since the concept of value prediction is entirely new, we cannot find many works dealing with this phenomenon. In fact, the only work that has also introduced this concept is by Lipatsi *et al.* in CMU ([Lipa96a] and [Lipa96b]). Our work and theirs, were independently developed in parallel, however they substantially differ from one another in their approach. While our work was originally motivated by the ambition to exceed the fundamental limits of ILP, Lipatsi *et al.* were

initially driven to employ value prediction in order to reduce the memory access time of load instructions (Lipa96a]). They attempted to predict outcome values of load instructions in order to hide the memory access time. In their further studies ([Lipa96b]), they have also recognized the tremendous potential of this phenomenon to boost the ILP beyond the dataflow graph limitations. Beyond these different approaches, the two works exhibit the further following differences (our future work is not included in the discussion):

1. Lipatsi *et al.* also revealed evidence in computer programs which strongly indicates that outcome values generated during the run-time of a program are likely to be predicted correctly. They termed this property "*value locality*". Value locality is actually very similar to a specific kind of value predictability that we termed "temporal value predictability". In addition, we have found that value locality is only "one side of the coin", since it is only a particular type of value predictability and it does not reveal all the value predictability properties that reside in a program. Beside value locality (or temporal value predictability) we have revealed that many programs also exhibit an additional kind of value predictability that we have termed "spatial value predictability". The potential of these concepts will be broadly discussed through the experiments that we will present in further sections.
2. We believe that the concept of value prediction is too premature to tailor it to an individual processor architecture. From our viewpoint, it is preferable to extensively explore the characteristics of this phenomenon before revising the design of current processors to take advantage of value prediction. In addition, we are also convinced that any attempt to progress in an opposite sequence may hide considerable knowledge that can be exploited in our further studies and may mask its extraordinary potential. Lipatsi *et al.* have chosen to revise the design of the PowerPC 620 ([Diep95]) and the ALPHA AXP 21164 ([Edmo95]) to exploit value prediction. We believe that their choice yielded limited performance gains because of the previously mentioned reasons. Instead, we have chosen to employ an abstract machine model which we consider to be very useful in our preliminary studies, since it provides us with the means to examine the pure potential of this phenomenon without being interrupted by the limitations of individual machines. Once the characteristics of this phenomenon are explored, we intend to examine its potential on practical machines and find how the design of current processors should be revised in order to gain the best advantages offered by this phenomenon.

In spite of the fact that both these works have been driven by different inspirations and approaches, we all recognize the extraordinary potential of this phenomenon to open new horizons in the research of program behaviors and microprocessor architectures.

The introduction of the notion of "value locality" by Lipatsi *et al.* has been motivated by the need to reduce memory latency and increase memory bandwidth. They have defined value locality as the likeliness of a previously-seen value to repeat itself within a storage location. Their measurements of the value locality for load instructions indicated that load instructions tend to reuse the previous values that they have generated. Their next step was to design a special mechanism, which they termed "Load Value Prediction Unit". This mechanism was attached to current processors model (PowerPC 620 and ALPHA AXP 21164), in order to enable them to exploit this phenomenon. The load value prediction unit consists of three components: 1. the load value prediction table, 2. the load classification table and 3. the constant verification table. The first component is a value predictor that operates in a very similar manner to the last-value predictor introduced in a previous section. The second component, load classification table, is responsible for classifying the load instructions

dynamically and deciding which load instructions are likely to be predicted correctly and which are not. The classification is important to avoid load value mispredictions and save the pay of extra penalty. In order to perform the classification they used a set of saturated counters each assigned to a set of load instructions. Each saturated counter is incremented when the prediction succeeds, and decremented otherwise. As a result the processor can attempt to decide according to the present state of the saturated counters which loads are likely to be predicted correctly and which loads are not. The present state of the saturated counters classifies the load into three categories: unpredictable, predictable and constant[*]. A value prediction of load instructions is only allowed for the last two categories. The measurements of Lipatsi *et al.* indicate that such a classification mechanism indeed works efficiently, but they neither provide an explanation of this phenomenon, nor examine what inherent properties of the program code that allows this classification mechanism to exploit them efficiently (these properties will be broadly discussed in Section 7 and Section 9). In order to verify the correctness of the load value prediction of the "predictable" class of loads, they simply compared the loaded value from memory against the predicted one. The verification of the "constant" class of loads, however, was handled by the third component, the constant verification unit. This unit allowed them to maintain the correctness verification of the "constant" class of predicted values without accessing the conventional memory system (cache, main memory etc.). This was enabled by forcing the entries in the load value prediction table, corresponding to the constant class to remain coherent with memory. The coherency was accomplished by invalidating the entries in the table each time that they were modified in memory. Their measurements indicate that approximately 10% of the loads can be considered "constant", however due to certain pipeline limitations inherent in the processors which they examined, they could only avoid accessing the memory systems in limited cases (cache banks conflicts). In addition they did not show how load value prediction competes with the traditional approaches of reducing memory latency such as prefetching ([Pint96]). We extensively study these issues in Section 8.

Lipatsi *et al.* have also recognized the extraordinary potential of value prediction to collapse true-data dependences ([Lipa96b]). They have also suggested predicting outcome values which have not been calculated yet, in order to exceed the serialization limits and execute true-data dependent instructions in parallel. In order to reach this goal, they have proposed extending their principle of value locality to cover not only load instructions but all instructions in the program generating outcome values. Their first step was to repeat the value locality measurements, this time referring not only to load instructions but to all the instructions that write values to registers. Their new measurements indicate that approximately 50% of the values that instructions generate, can be correctly predicted by employing a very similar mechanism to the last-value predictor (Section 4). Their next step was to extend the load value prediction unit to a more extensive scheme that could exploit value locality not only from load instructions, but also from all the instructions in the program that produce output values. This scheme is similar to the previous one and also consists of three units: the value prediction table, the classification table and the verification unit. The first two units are in fact extensions of the previous load value prediction table and the load classification table. These units attempt to predict and classify all the instructions that write values to registers and not only load instructions. The third unit is responsible for verifying whether the value prediction was correct or not. In order to measure the performance gain that this mechanism can accomplish, Lipatsi *et al.* have chosen to

[*] The name of this class does not imply that the loads that belongs to it truly generate constant values.

implement it on three microprocessors models. Two of these models are based on the PowerPC 620, and the third model is an abstract model that *takes into account the major limitations of current processors* (such as: branch misprediction penalty, limited instruction fetch bandwidth etc.). In their processor models, each time a dependent instruction was executed and one or more of its source operands was fed with an incorrect predicted value, they needed to pay an extra cycle as a penalty in order to recover from the value misprediction. This penalty occurs since their machine models need an extra cycle to verify the value prediction beyond the pipeline latency of the computed result. In addition an incorrect value prediction can cause a structural conflict (hazard) since a mispredicted instruction prevents other useful instructions from being executed by occupying machine resources for an additional clock cycle (that is needed for the verification). Their measurements indicate, however, that most of the value mispredictions are avoided since the classification unit is efficient enough to eliminate them. Structural conflicts may also appear even when values are predicted correctly since instructions that were executed based on speculative values are not allowed to evacuate the reservation stations (instruction window) until their predicted input values are verified. The performance measurements of Lipatsi *et al.* indicate that value prediction gains an average speedup that varies from approximately 5% to more than 10% (depending on the configuration of the value prediction scheme) in their practical machine models. In addition, they are also aware that a significant portion of the performance gain that value prediction can accomplish is masked by the limitation of current processor architectures, such as the limited bandwidth of the instruction fetch, the branch prediction accuracy etc.

Another prior work that also has a certain relation to this work is the *value cache* in the *TM architecture* ([Harb82]). This hardware mechanism is used in order to eliminate redundant computation that appears in common sub-expressions. The value cache stores result values of phrases in order to avoid their recomputation. The use of this mechanism is limited since it should be guaranteed that the variables in the phrase were not changed since the latest computation of the phrase that is stored in the value cache. When one of the variables of a phrase is modified, the corresponding entry in the value cache needs to be invalidated. The nature of the value cache is not speculative, i.e. it does not predict values. Once this mechanism finds a matching phrase and provides the value that the phrase generates, there is no need to verify the correctness of the provided value. In addition, value cache cannot really go beyond dataflow dependences, it can only save computation time of expressions that have been previously computed.

6. Simulation environment

A special trace driven simulator was developed in order to provide measurements for the experiments that are presented in the following sections. The simulation environment was fed with the Spec95 benchmarks suit (table 6.1). The benchmarks were traced by the SHADE trace generator simulator [SHADE] on Sun-Sparc microprocessor. All benchmarks were compiled with the GCC 2.7.2 compiler with *all available optimizations*.

SPEC95 Benchmarks		
Benchmarks	Type	Description
go	Integer	Game playing.
m88ksim	Integer	A simulator for the 88100 processor.
gcc1, gcc2	Integer	A C compiler based on GNU C compiler version 2.5.3 compiling 2 different input files.
Compress95	Integer	Data compression program using adaptive Lempel-Ziv coding.
li	Integer	Lisp interpreter.
ijpeg	Integer	JPEG encoder.
perl1, perl2	Integer	Anagram search program with two different input files.
vortex	Integer	A single-user object-oriented database transaction benchmark.
tomcatv	FP	A vectorized mesh generation program.
swim	FP	Shallow water model with 1024 x 1024 grid.
su2cor	FP	Quantum physics computation of elementary particles masses.
hydro2d	FP	Hydrodynamical Navier Stokes equations solver to compute galactical jets.
mgrid	FP	Multi-grid solver in computing a three dimensional potential field.

Table 6.1 - The spec95 benchmarks.

The set of benchmarks that were used consisted of 8 integer benchmarks and 5 floating-point benchmarks. Each integer benchmark was traced for 100 millions of instructions. In addition, two of the integer benchmarks, gcc and perl, were examined by using two different input files in order to obtain a certain evaluation about the effect of the input file on the characteristic of values predictability. The floating point benchmarks (except mgrid) consist of two major execution phases: an initialization phase and a computation phase. In the initialization phase the data is read from input files, while the computation phase performs the actual computation. Hence the corresponding experimental results refer to both these phases respectively. The initialization phase was traced till it was completed (the instruction count is in the order of hundreds of millions of instructions) and the computation phase was traced for 100 millions of instructions.

7. Characterization of value predictability

This section presents results of various experiments that have been made in this research. Primarily, substantial evidence is provided proving that programs exhibit remarkable potential of value predictability. In addition a broad study of various aspects and characteristic of this phenomenon is presented.

7. 1. Value prediction accuracy

The potential of the concept of value prediction significantly depends on the prediction accuracy that it can accomplish. There are two different fundamental components which determine the value prediction accuracy: (1) the value predictor scheme itself and (2) the potential of value predictability that resides within the program code. The first component is related to the structure of the predictor and its capabilities that eventually determine the prediction formula. The second component, however, is independent from the predictor organization since it reflects inherent properties of the program and its databases. In the first place, our experiments will provide substantial evidence which confirms that programs tend to exhibit value predictability. In addition, they will also focus on the relations between these two components by examining how efficiently various predictors exploit the different value predictability patterns (temporal value predictability and spatial value predictability) that programs exhibit. It is important to clarify again that at this stage of the research our aim is not to argue any particular hardware implementation of the predictor schemes. Rather, our initial purpose is to focus on studying the phenomenon from as general as possible perspective. In addition, we are also convinced that in order to better integrate the concept of value prediction to superscalar processors it is preferable to first accumulate the substantial knowledge about this phenomenon and only then combine the hardware consideration.

Our experiments evaluate four value predictors: the *last-value* predictor, the *stride* predictor, the *SEF* predictor (only for floating point instructions) and the *register-file* that were all described in Section 4. Due to our abstract perspective, the prediction table size of the first three predictors is considered to be unlimited in the experiments. The programs that our simulations examine include both integer and floating-point Spec95 benchmarks. Some of these programs will even be examined with different input files in order to evaluate the significance of the program's database to its value predictability. In the integer benchmarks the prediction accuracy is measured separately for two sets of instructions, load instructions and ALU (arithmetic-logic) instructions. In the floating-point benchmarks the prediction accuracy is measured for two *additional* sets of instructions: floating-point load instructions and floating-point computation instructions.

The first set of measurements consists of the value prediction accuracy of each of the value predictors for integer load instructions in the integer benchmark, as illustrated by figure 7.1.

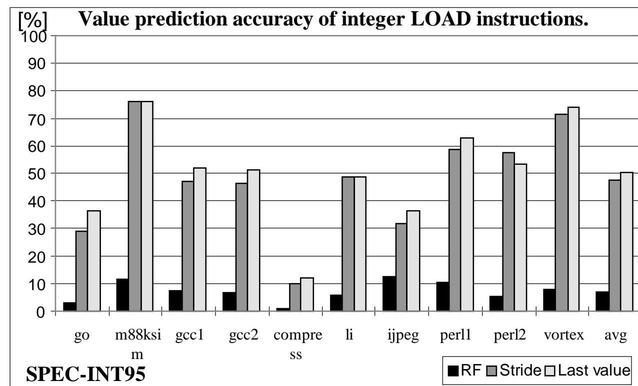

Figure 7.1 - Value prediction accuracy of integer load instructions in Spec-Int95.

This figure illustrates remarkable results, since it indicates that nearly 50% (on average) of the values that are generated by load instructions can be correctly predicted by two of the proposed predictors, the last-value predictor and the stride predictor. It can be noticed that the value prediction accuracy of the stride predictor and the last-value predictor is quite similar in all the benchmarks, indicating that these integer load instructions barely exhibit spatial value predictability consisting of stride patterns. This implies that for this type of instructions in integer programs the cost of extra stride field in the prediction table of the stride-predictor does not seem to be attractive. The prediction accuracy of the integer load does not spread uniformly among the integer benchmarks. It can noticed that in some benchmarks the load values are relatively highly predictable, like the benchmarks *m88ksim* and *vortex*, where the prediction accuracy of both last-value and stride predictors is relatively high (more than 70%), while the prediction accuracy of other benchmarks, like the *compress* benchmark, is relatively very poor (about 10%). In all the benchmarks, the register-file predictor yields a relatively poor prediction accuracy (less than 10% on average) since it can hardly exploit any kind of value locality.

Figure 7.2 shows more outstanding results about the prediction accuracy which the value predictors gain for ALU instructions in integer applications:

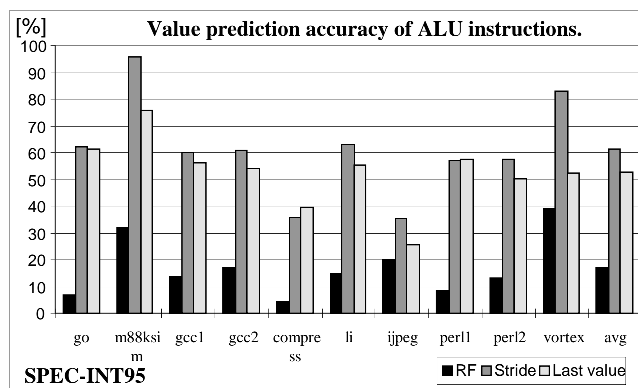

Figure 7.2 - Value prediction accuracy of ALU instructions in Spec-Int95.

First, these experiments provide additional encouraging evidence about our capability to predict outcome values. They indicate that a very significant portion of the values that are generated by the ALU instructions are likely to be predicted correctly by our value predictor schemes. In the average case, the stride predictor gains a prediction accuracy of 61%

compared to the last-value predictor that gains only 52%. In several benchmarks, like go and perl, the last-value predictor and the stride predictor gain similar value prediction accuracy. Beyond the temporal value predictability that these programs exhibit, they do not exhibit spatial value predictability consisting of strides, and therefore both predictors gain similar value prediction accuracy. In these cases, it is expected that most of the correct value predictions of the stride predictor are accomplished by stride values that are actually zero. This expectation will be verified in few more subsections. While the only pattern that our predictors could exploit from load instructions is temporal value predictability, ALU instructions also exhibit in several benchmarks, like the m88ksim and vortex, a considerable potential of spatial value predictability that consists of stride patterns. This kind of value predictability can only be exploited by the stride predictor beyond the temporal value predictability that can be exploited by both last-value and stride predictors. This observation is expressed in the significant gap between the value prediction accuracy of the stride predictor in comparison to the last-value predictor. Hence, in those benchmarks which also exhibit spatial value predictability it is expected that the contribution of non-zero strides to the correct value prediction in the stride predictor will be more significant compared to the previous benchmarks. This expectation will be verified as well. As in the previous case, the register-file predictor yields relatively poor prediction accuracy compared to the other predictors. Its prediction accuracy varies from 4.2% in the benchmark compress to 38.96% in vortex, yielding an average value prediction accuracy of nearly 17%. The overall prediction accuracy for load and ALU instructions is summarized in the following table for the three predictors schemes:

Spec95 integer benchmarks			
Overall prediction accuracy [%]			
Benchmark	register-file	stride	last-value
go	5.72	52.27	53.82
m88ksim	28.55	92.98	77.94
gcc1	11.54	55.76	55.20
gcc2	13.46	56.51	54.05
compress95	3.59	28.48	31.41
li	11.57	58.11	54.28
ijpeg	18.46	35.48	28.95
perl1	9.33	60.41	61.76
perl2	10.27	57.95	52.68
vortex	31.14	78.67	58.16
average	14.36	57.66	52.82

Table 7.1 - Summary of overall prediction accuracy.

An additional important *preliminary* observation, indicates that different input files do not dramatically affect the prediction accuracy of the benchmarks as illustrated for the gcc (gcc1 and gcc2) and the perl (perl1 and perl2) benchmarks. This property has a tremendous significance when considering the involvement of the compiler to boost value prediction

accuracy. One of the future directions considered in this work is the use of a program profiling in order to increase the prediction accuracy by adding to the machine's instruction-set a set of compiler directives in order to assist the hardware to determine which instructions are likely to be predicted correctly. This may indicate that the information collected by the compiler during the profiling phase (running the application with training input files) can be significantly correlated to the true situation where the application runs its real input files.

The next set of experiments presents value prediction accuracy in *floating point benchmarks*. The prediction accuracy is measured in each benchmark for two execution phases*: initialization (denoted by #1) and computation (denoted by #2). First the prediction accuracy is measured for integer instructions (load an ALU) and afterwards the prediction accuracy for the floating point instructions will be presented as well. Figure 7.3 exhibits the value prediction accuracy for integer loads in floating point benchmarks. It reveals that unlike the corresponding case in the integer benchmarks, in this case the floating point benchmarks exhibit spatial value predictability which consists of stride patterns. These patterns are exploited by the stride predictor that gains an average accuracy of 70% in the initialization phase and 63% in the computation phase, in comparison to the last-value predictor that gains an average accuracy of nearly 66% in the first phase and only 37% in the second. The reasons for the significant prediction accuracy gap between these predictors in computation phase, will be explained in a further discussion in this work. We also notice that as in the previous cases, the register-file predictor gains a relatively poor value prediction accuracy of only 2-4%.

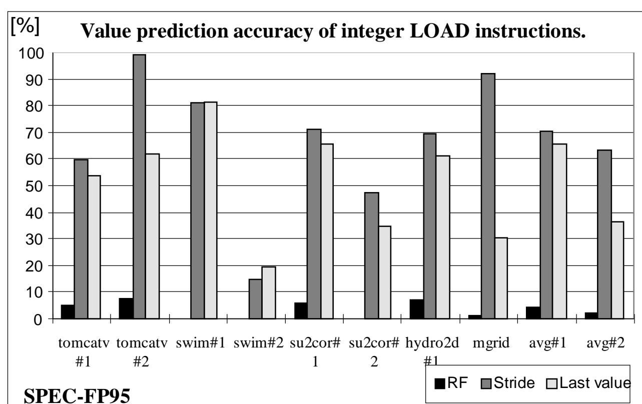

Figure 7.3 - Value prediction accuracy of integer load instructions in Spec-FP95.

When the prediction accuracy is examined for ALU instructions in the floating point benchmarks, it reveals several more interesting results (figure 7.4). In the initialization phase, the three predictors do not exhibit exceptional behavior in comparison to the integer benchmarks. However, in the computation phase of all the floating point benchmarks, the gap between the prediction accuracy of the last-value predictor and prediction accuracy of the stride predictor becomes very significant. In the computation phase most of the ALU instructions exhibit stride patterns rather than repeating their recently generated values, and therefore the stride predictor can take advantage of this pattern of value predictability. The stride predictor gains in the computation phase average prediction accuracy of 95%, while the last-value predictor gains only 23%.

* except the benchmark mgrid where the initialization phase is relatively negligible.

We will discuss in detail the reasons for this observation in a later subsection. In addition, unlike previous cases where the register-file predictor gained relatively poor value prediction accuracy, in this case it gains competitive prediction accuracy of nearly 65% that even outperforms the last-value predictor.

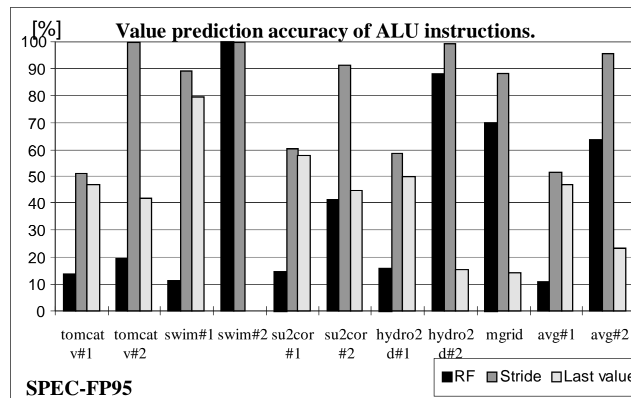

Figure 7.4 - Value prediction accuracy of ALU instructions in Spec-FP95.

The next two figures exhibit the prediction accuracy for two kinds of floating point instructions, floating point loads and floating point computation instructions. They illustrate that our three value predictors can hardly gain significant prediction accuracy in these instructions, since floating point values show neither temporal value predictability nor spatial value predictability (based on stride patterns) as indicated by figures 7.5 and 7.6. In floating-point loads the average value prediction accuracy of the last-value predictor and the stride predictor is more than 40%, and in the floating point computation instructions they gain less than 30% of an average prediction accuracy. One of the reasons that may explain why it is harder to predict floating values with these predictors is the representation of these values. Floating point values are represented by three value fields: sign, exponent and fraction (mentisa). It is hard to expect these value predictors, that by their nature tend to fit prediction of integer values, to successfully perform prediction of floating point values. In adidition, floating point computations are considerably less trivial than integer computations, making them hard to be predicted. This can also explain why floating point loads exhibit more value predictability in comparison to the floating point computation, since one can expect to find more patterns of regularity and reuse of values in floating point loads rather than floating point computations.

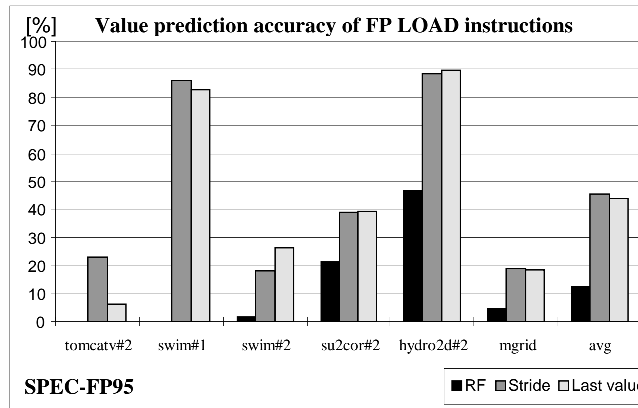

Figure 7.5 - Value prediction accuracy of floating point loads.

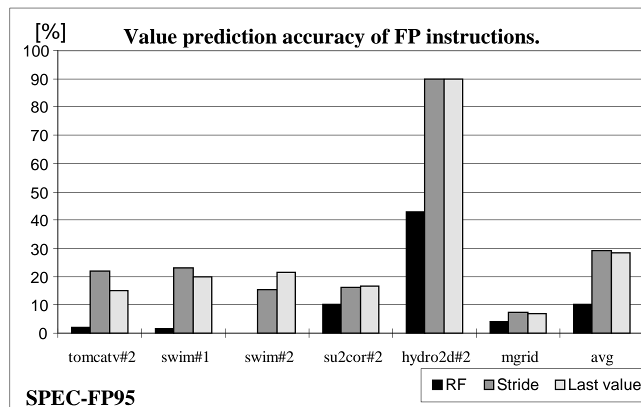

Figure 7.6 - Value prediction accuracy of floating point computation instructions.

The relatively low prediction accuracy of floating point values has motivated us to explore other methods of value prediction for such data types. A possible method that was described at Section 4 is the usage of the SEF predictor. Unlike the approach of the previous predictors that have attempted to predict the entire outcome value, this predictor scheme attempts to predict the sign, exponent and fraction of floating point values separately. Figures 7.7 and 7.8 illustrate the prediction accuracy gained by the SEF predictor for the sign ,exponent and fraction values of floating point loads and floating point computations respectively. These results show that the sign and exponent values are significantly more likely to be correctly predicted than the fraction value. The sign's average prediction accuracy is approximately 90% and the exponent's average prediction accuracy is around 70% for both sets of instructions. In addition, these results are consistent with the results of figures 7.5 and 7.6 which indicated that values that were generated by floating point load instructions are more likely to be correctly predicted than those generated by floating point computations.

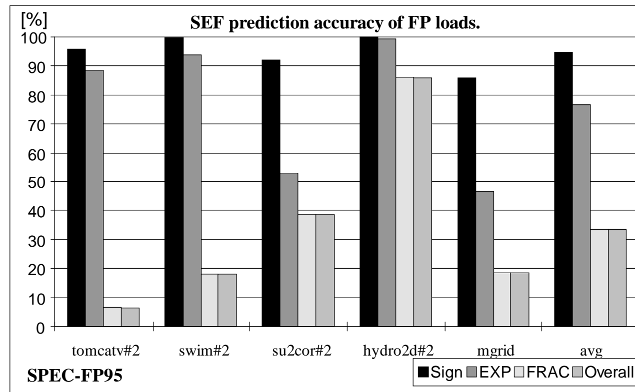

Figure 7.7 - SEF prediction accuracy of floating point loads.

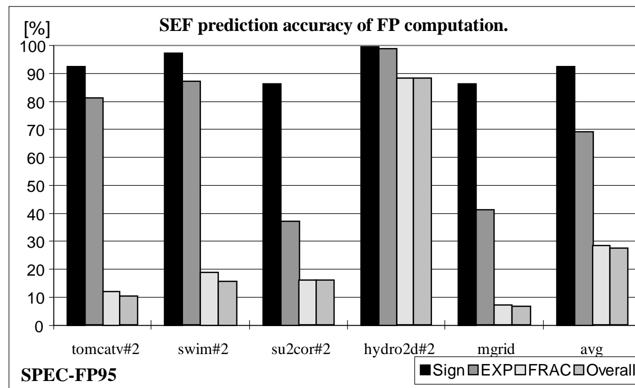

Figure 7.8 - SEF prediction accuracy of floating point computations.

These observations can have significant implications since they motivate us to compromise. Instead of predicting the entire floating point value with relatively poor prediction accuracy, it is suggested to focus only on the highly predictable portions of the floating point value. This approach is termed *partial value prediction*. An instant question that may arise when discussing partial prediction of floating point values is what is its practical usage? Since floating point operations are quite complicated, their execution may take several processor cycles. It is implied that certain floating point operations may start their execution even if only a portion of the entire information they need is available. For instance: floating point add instruction, initially compares the exponent values of the source operands. Hence, supplying predicted partial values that are required for the initial computation, may decrease the effective time that instructions should wait for their operands to be available. Still, the concept of partial value prediction that seems quite attractive should be studied and evaluated more broadly in a future study.

7. 2. Distribution of value prediction accuracy

The value prediction accuracy that was measured in a previous subsection is an average number that it is important in order to evaluate the performance of the predictors. However this average number does not provide information about distribution of the prediction accuracy among the instructions in the program. The following measurements will attempt to

provide a deeper study of the statistical characteristics of value prediction by examining the distribution of value prediction accuracy among the instructions in the program. In addition, we shall also discuss how this knowledge can be taken advantage of.

Figure 7.9 illustrates the distribution of value prediction accuracy of the stride predictor among the instructions in the program (referring only to the value-generating instructions that the predictor has collected during run-time). It indicates that the prediction accuracy does not spread uniformly among the instructions in the program. Approximately more than 40% of the instructions (value-generating instructions) are very likely to be correctly predicted with prediction accuracy greater than 70%. In addition, approximately the same number of instructions are very unlikely to be correctly predicted with a prediction accuracy less than 40%.

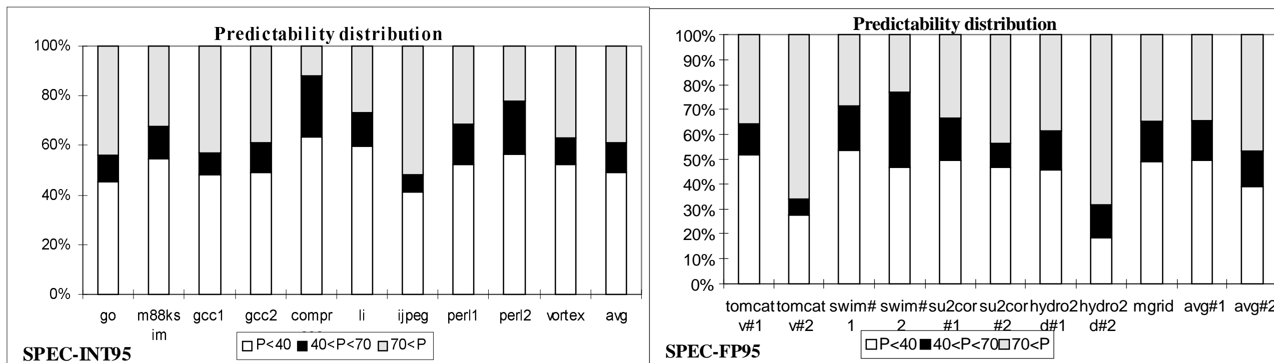

Figure 7.9 - Predictability distribution of values.

The consequences of these results are very important, since they motivate us to develop mechanisms that will allow us to distinguish between the predictable and unpredictable instructions and filter the unpredictable ones. The contribution of such classification can be very significant to each of these aspects:

1. The classification can significantly increase the effective value prediction accuracy of the predictor by eliminating (filtering) the value prediction of the unlikely predictable instructions. A preliminary study of the effect of such classification on the general performance will be discussed in later section of this work.
2. The replacement mechanism of the prediction table can now exploit this classification and prioritize the entries to be replaced more efficiently. Eventually, this may significantly increase the effective utilization of the prediction table.
3. In certain microprocessor architectures miss-predicted values may cause some extra miss prediction penalty due to their pipeline organization. Therefore by classifying the instructions, the processor may refrain from predicting values from the class of unpredictable instructions and save the miss-prediction penalty.

Two possible methods can be considered for classification:

1. A possible method to classify instructions is the use of saturated counters (figure 7.10). An individual saturated counter is assigned to each entry in the prediction table. At each occurrence of successful prediction the counter is incremented, or decremented conversely. Upon the present state of the saturated counter, the processor can decide whether to consider the suggested prediction or to avoid it. A possible state transition chart of such a saturated counter is illustrated in figure 7.10. Such a method for value prediction classification was suggested by Lipatsi *et al.* in [Lipa96a].

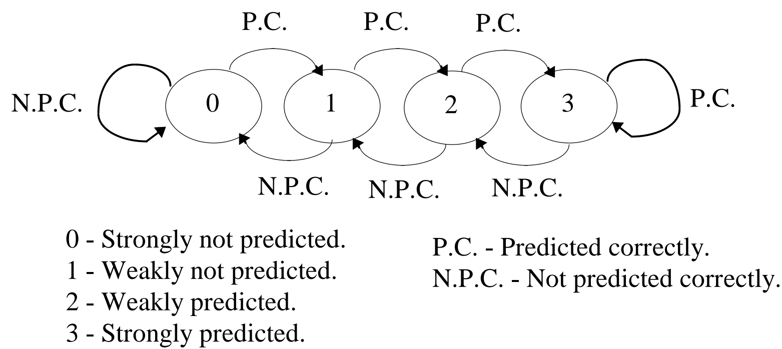

Figure 7.10 - 2-bit saturated counter for value prediction filtering.

2. Another method that we consider for classification and filtering of unpredictable instructions is the use of compiler profiling technology. The compiler can collect information about the value predictability of instructions according to previous runs of the program. Then, it can place directives in the opcode of the instructions providing hints to the processor for classifying the instructions. From previous subsection we recall that the measurement of the expected correlation of value prediction accuracy between different runs of the program with different input files are encouraging. This direction is intended to be broadly explored in our future work.

7. 3. Non-zero strides and immediate operations

The stride predictor extends the capabilities of the last-value predictor since it can exploit both temporal and spatial value predictability consisting of strides. In this subsection we examine how efficiently the stride predictor takes advantage of its additional stride fields beyond the last-value predictor to exploit stride value patterns. We consider the additional stride fields to work efficiently only when the predictor accomplishes correct value predictions that are based on non-zero strides. In the following measurements we examine the number of successful predictions that were based on non-zero strides out of the overall number of correct predictions. These measurements are presented in table 7.2. In the integer benchmarks the average ratio of the successful zero-stride based predictions out of the overall successful predictions is more than 16%, and in the floating point benchmarks it varies from 12% in the initialization phase to 43% in the computation phase. The relatively high ratio of non-zero strides in the floating point computation phase is explained by the significant contribution of immediate integer add and subtract instructions to the successful predictions.

Ratio of successful non-zero stride-based predictions out of overall successful predictions.			
Spec95 integer		**Spec95 floating point**	
Benchmark	[%]	Benchmark	[%]
go	8.83	tomcatv#1	13.93
m88ksim	16.42	tomcatv#2	55.14
gcc1	12.79	swim#1	8.98
gcc2	15.44	swim#2	65.9
compress95	6.52	su2cor#1	9.35
li	15.22	su2cor#2	27.74
ijpeg	36.37	hydro2d#1	16.28
perl1	7.57	hydro2d#2	15.09
perl2	15.27	mgrid	51.4
vortex	30.34	**average#1**	**12.14**
average	**16.48**	**average#2**	**43.06**

Table 7.2 - The distribution of non-zero strides.

This table, however, may lead the reader to an incorrect conclusion about the effectiveness of the stride predictor to exploit non-zero strides and its significance to the expected ILP improvement. For instance, it shows that 16.4% out of successful predictions in the benchmarks m88ksim are because of non-zero strides and 15.2% in the benchmark li. Does this mean that the contribution of the stride predictor and non-zero strides to these two benchmarks is the same? - Obviously not; one should be aware of the fact that these results should be given the appropriate weight, i.e. their relative number of appearances in the program's execution. Moreover, the connection between the prediction accuracy and the expected boost in ILP is not straightforward since the distribution of its contribution may not be uniform. If the importance of non-zero strides is crucial to the ILP of the application, then even an improvement of approximately 10% in the prediction accuracy can be very valuable. Therefore figure 7.11 can provide a better illustration, yet not complete, of the significance of both non-zero and zero strides. This figure presents the overall successful prediction based on non-zero strides out of the overall instruction count. It can be noticed that the benchmarks m88ksim, ijpeg and vortex exhibit greater tendency of non-zero stride values compared to the rest of the integer benchmarks. Still, this figure may not absolutely reflect the real contribution of non-zero strides, since we did not measure yet the relative contribution of these successful non-zero stride predictions to the ILP. These measurements will be presented in section 9. The current knowledge about the characteristics of non-zero strides may motivate us in the future to explore the usage of hybrid predictors, that combine both the last-value predictor and the stride predictor, where the last-value predictor would be dedicated for zero stride (reuse of value) and the stride predictor would be dedicated for non-zero strides.

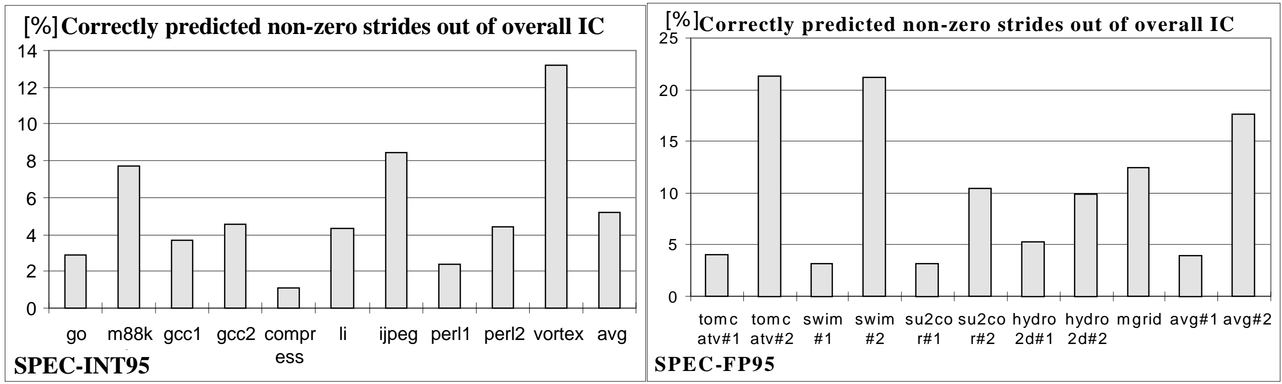

Figure 7.11 - The distribution of zero-strides out of overall instruction count.

Non-zero strides seem to appear for various reasons, e.g. immediate add and subtract instructions, and computations of addresses of memory references that step with a fixed stride on arrays in memory. Figure 7.12 illustrates the contribution of immediate add and subtract instructions to the overall number of successful predictions in the stride predictor. In the integer benchmarks it is nearly 20% (on average) and in the floating point benchmarks it varies from nearly 15% in the initialization phase to more than 30% in the computation phase (on average). The significant gap between the contributions in the initialization phase and in the computation phase of the floating point benchmarks can be explained by the fact that most memory accesses of floating point benchmarks consist of stride patterns ([Pint96]). When the portion of the successful value predictions that serve address calculations of memory pointers (for data items only) is examined, it reveals that this number is considerably more significant in the computation phase than the initialization phase as illustrated by figure 7.13. Section 8 confirms this observation as well.

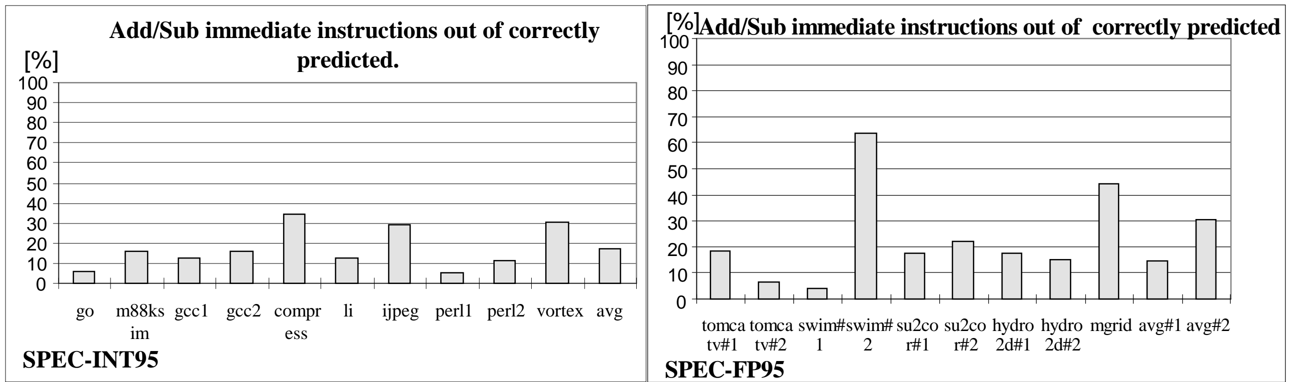

Figure 7.12 - The contribution of Add/Sub immediate instructions to the successful predictions.

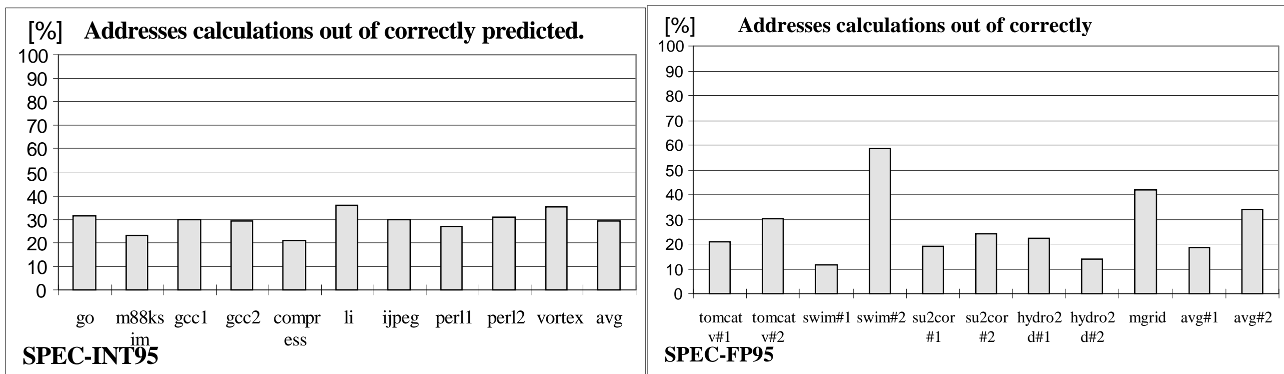

Figure 7.13 - Addresses calculations out of successful predictions.

We end this section with the following conclusion:

- We have provided substantial evidence confirming that programs tend to exhibit two kinds of value predictability, temporal and spatial, and examined how various predictor schemes exploit these properties.
- When we examined the value prediction accuracy, we have distinguished between four types of instructions: integer loads, ALU, floating point loads and floating point computations instructions. Our measurements indicate that:
  - Integer loads in integer benchmark mostly exhibit temporal value predictability and barely spatial value predictability in form of strides. In the floating point benchmarks the integer loads behave similarly in the initialization phase, but in the computation phase they also significantly exhibit spatial value predictability that consists of strides value patterns. In the integer benchmarks the use of the stride predictor to predict the value of loads does not introduce an additional advantage since it accomplishes a similar prediction accuracy to the last-value predictor (approximately 50%). In the floating point benchmarks, however, the most noticeable gap between these predictors appeared in the computation phase where the stride predictor gained prediction accuracy of 63% while the last-value predictor accomplished only 37%.
  - ALU instructions exhibit both temporal value predictability and spatial value predictability that consists of stride patterns. In the integer benchmarks the stride predictor improves the prediction accuracy of the last-value predictor from 50% to 60%. In the computation phase of floating point programs, ALU instructions are dominated by spatial value predictability that consist of strides. The stride predictor gained approximately prediction accuracy of 95% while the last-value predictor only gained 23%. In these benchmarks we observed that even the register file predictor can be a competitive scheme since it accomplished 65% prediction accuracy.
  - Floating point values barely show neither last-value locality nor stride-based value locality since they are less triviail to be predicted. In both floating point loads and floating point computation instructions our predictor schemes gained relatively poor prediction accuracy. In addition, we have found that the highly predictable parts in the floating point value are the sign and exponent. We have indicated that this observation may motivate us in the future to employ partial value prediction for these instructions.

- Our studies of the stride patterns indicate that approximately 85% out of the correct value predictions were based on zero strides and the rest on non-zero strides. This observation motivates us to develop a hybrid prediction table that combines both the stride predictor and the last-value predictor. The combination of these schemes would allow us to better utilize the extra stride field that would be dedicated only to the instructions that exhibit stride patterns.
- We have shown that the prediction accuracy does not distribute uniformly among the instructions in a program. Most programs exhibit two sets of instructions, highly value-predictable instructions and highly unlikely-predictable ones. This observation motivates us to develop classification mechanisms that could filter out the unlikely predictable instructions form being candidates for value prediction. In addition, the classification can better utilize the prediction table, increase the effective prediction accuracy of the predictor and eliminate value mispredictions. Beyond the usage of staurated counter for the classification, we consider an additional classification scheme. This scheme is based on compiler profiling that would collect information about the tendency of instructions to be predictable according to previous runs of the program, The compiler can pass to the processor hints through the opcode of instructions about the tendency to be predictable or not.
- Our preliminary observation indicates that different input files do not significantly change the value predictability of a program. This observation is encouraging about our future intention to use profiling-based compiler techniques that could classify the value predictability of instructions based on previous runs of the programs.

8. Characteristics of value predictability in load instructions

In this section we focus on the characteristics that the concept value prediction exhibits in load instructions. Load instructions may have a significant impact on the program performance from two major aspects: 1. they may cause the processor to stall due to memory system latencies and 2. they may cause true-data dependences as other value-generating instructions, since the value that they read from memory may be consumed by other instructions. Our main purpose is to examine how the concept of value prediction can affect and support these two aspects.

In order to explore the contribution of value prediction from the first aspect, we examine all the load memory references that are involved in cache misses and their loaded memory values. First, we are interested in studying what portion of these loaded memory values is "value-predictable" and what the relations are between the *memory address predictability* and the value predictability of these references.

Definition 25: ***Address predictability*** - is the potential to predict *memory addresses* generated by memory reference instructions (e.g. load or store instructions) before they are executed. This property is inherent to the program and depends on the memory locality patterns that the program exhibits.

Address predictability of memory references is determined by the memory locality patterns of the program. This property was widely exploited by different data prefetching mechanisms in order to reduce the average memory access time ([Pint96]). Such mechanisms employ special predictors that allow them to predict the future addresses of memory references and fetch them before they are demanded by the program. We term these predictors *address predictors*:

Definition 26: ***Address predictor*** - is a hardware-based mechanism that attempts to predict future memory references of a program.

Examining the value predictability patterns in the load cache misses and comparing them to the address predictability patterns can provide valuable information for the following questions:

1. What is the effectiveness of value prediction to reduce the penalty of load cache misses by attempting to predict their values (as it was proposed by Lipatsi *et al.* in [Lipa96a])?
2. How successfully can value prediction compete with other techniques such as data prefetching?

Our following experiments will attempt to provide answers to these questions by focusing on the potential and the patterns that programs exhibit and how it is exploited by the value predictor and the address predictor.

The structure of the competitive address predictor scheme that we use in our experiments is similar to our value predictor schemes. It is organized as a table (for instance a cache table) with an unlimited number of entries, where each entry is assigned uniquely to a previously-seen load instruction. Each entry contains three fields: *Tag* - identifies the individual load instruction according to its instruction address, *Last address* - the address of the last cache block that was fetched by the load instruction and an *Address stride* - that is determined according to the difference between two consecutive memory accesses of each of the individual loads. The predicted address that the prefetching scheme fetches is determined according to the last address field plus the stride field. The chosen value predictor scheme to compete the address predictor is the last-value predictor, since it has gained best prediction accuracy for load instructions compared to our value predictor schemes. The data cache parameters that were chosen for the experiments are quite typical to common modern microprocessors (PowerPC™, Pentium™, PentiumPro™): 16 KB size, 4-way set associative and 32 bytes line size.

Figure 8.1 illustrates the correlation between the *address prediction accuracy* and value prediction accuracy out of the load misses.

Definition 27: ***Address prediction accuracy*** - is the number of successful address predictions out of the overall number of address prediction attempts. The factors that determine address prediction accuracy are: (1) the address predictor scheme and its capabilities (resources etc.) and (2) address predictability of the program.

This figure exhibits four possible sets of load references: 1. load references that *both their values and addresses* can be predicted correctly, 2. load references that *only their addresses* can be predicted correctly , 3. load references that *only their values* can be predicted correctly and 4. load references that *neither their values nor their addresses* can be predicted correctly. It can be noticed that out of the load misses in the integer benchmarks, the portion of the third set (values only) is competitive to the portion of the second set (addresses only). Therefore, in these benchmarks, value prediction can contribute significantly by handling a considerable set of load references that cannot be managed even by data prefetching. However, it can be seen that in the floating point benchmarks, most of the load cache misses that exhibit correctly predicted values, necessarily exhibit correctly predicted addresses, i.e., the portion of the third set is negligible. Figure 8.2 is similar to figure 8.1 - it illustrates the correlation between the address prediction accuracy and value prediction accuracy when the bars of each benchmark are given the appropriate weight according to the load miss rate.

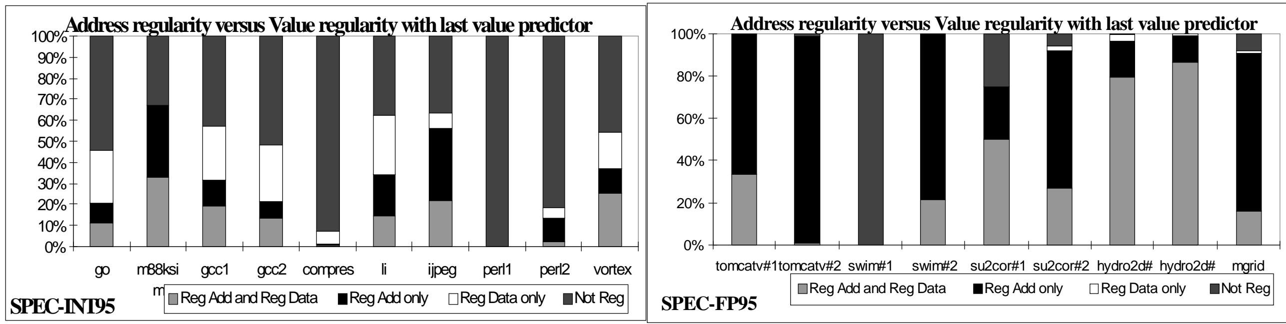

Figure 8.1 - Address regularity versus value regularity out of the overall load misses.

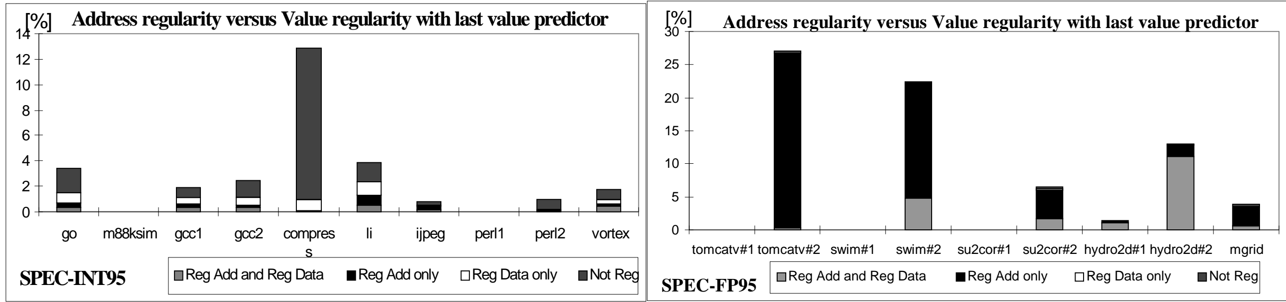

Figure 8.2 - Address regularity versus value regularity.

In order to address the second aspect, we measure the ILP that can be exploited when applying value prediction only for load instructions. It is assumed in the following experiments that the memory system is perfect (memory references never stall the processor) in order to eliminate the contribution of the first aspect (memory latency). In addition, in order to avoid discussing yet individual implementation issues, an abstract machine is considered with an unlimited number of execution units and physical registers, but a restricted instruction window size. Each instruction is assumed to take only a single cycle. The ILP that such a machine can exploit is limited by the dataflow graph of the program and by the size of the instruction window. Figure 8.3 shows the ILP that can be gained by employing the last-value predictor in comparison to a machine that does not employ value prediction. It indicates that in some benchmarks like m88ksim, li and perl the contribution of load value prediction is significant while in some other benchmarks like compress, vortex and mgrid it is barely noticeable. These variations are highly dependent on the value predictability patterns that these programs exhibit and their contribution, in particular to the value predictability patterns of the load instructions. In the following section we shall present the ILP gained by employing value prediction to the entire set of value-generating instructions (not only loads), so it will help us to evaluate the significance of the contribution of load value prediction to the ILP as well.

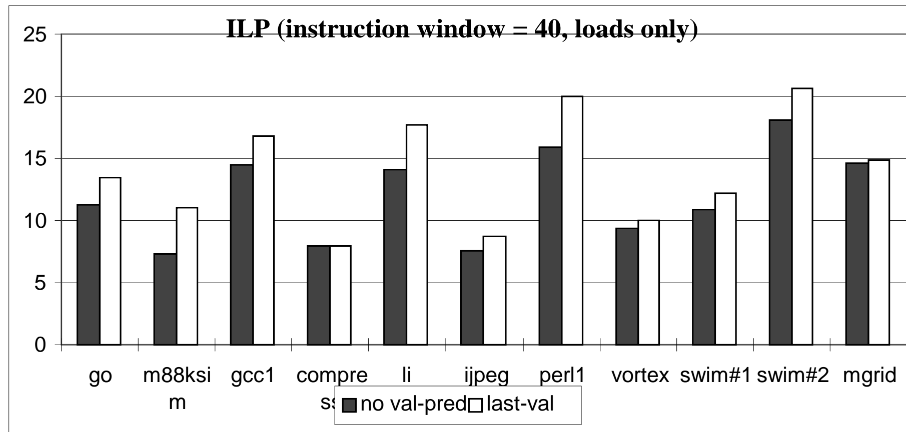

Figure 8.3 - The comparable ILP of loads value prediction.

We end this section with the following conclusions:

- Load instructions in integer programs can gain significant benefit from value prediction in order to improve the performance of the memory system beyond the conventional prefetching techniques. However, in the floating point benchmarks, all the value-predictable loads necessarily exhibit address predictability, and therefore they all can be handled by data prefetching mechanisms.
- When the effect of the memory system is not considered in our experiments, the value prediction of (only) load instructions introduces a noticeable gain in the ILP however it is limited since load instructions may not dominate the parallelism of the program dramatically. For instance, in many cases loop-carried dependences are dominated by ALU instructions that control the indexes calculations of the loops, as was observed in our sample program in Section 2 and Section 3.

9. The impact of value prediction on ILP

The ILP that a processor can exploit out of a serial program is an indicator of the amount of resources that it can utilize efficiently. Obviously, it makes no sense to employ extensive resources when the ILP that the processor can extract from the program is very small. This section will present the effect of value prediction on the ILP that can be exploited in various benchmarks. Since the concept of value prediction is entirely new and in order to avoid discussing yet particular implementation issues inherent to different processor architectures, our experiments consider an abstract machine with a finite instruction window. The ILP which such a machine can gain (when it does not employ value prediction) is dictated by the dataflow graph of the program and its instruction window size. We have previously indicated that in order to reach the dataflow graph boundaries, a machine should employ: 1. unlimited number of resources (execution units etc.), 2. unlimited number of registers, 3. perfect (either static or dynamic) branch prediction mechanisms and 4. the throughput of the instruction fetcher should be sufficient. In addition, we also assume, for simplicity, that each instruction can be executed in a single cycle. The abstract machine model considered in our experiments is very useful to the preliminary studies of value prediction, since it provides us with a means to examine the pure potential of this phenomenon without being interrupted by individual machines limitations. Once the characteristics of this phenomenon are explored, we shall examine its potential on

practical machines and try to study the performance gap and how to reduce it. As a part of our abstract perspective, we also assume that there is no extra penalty when values are not predicted correctly, since we believe that the commit procedure of the predicted values needs to be further studied in the future work of our research. In addition it will be shown in this section that most of the value mispredictions can be eliminated by employing filtering mechanism with almost no cost to the exploited ILP. The following experiments consist of two sets: the first set measures the effect of various value prediction policies and predictor schemes on the ILP under two different instruction window sizes, and the second set of measurement presents the effect of a filtering scheme on the ILP that value prediction can exploit.

9. 1. Various value prediction policies and the impact of the instruction window size on ILP

In previous sections we have broadly studied the value prediction accuracy that various value predictors schemes can gain. It is important to indicate that the connection between the value prediction accuracy gained by these predictors and the expected boost in the ILP may not be straightforward. It is not sufficient that these schemes can correctly predict outcome values, these predictable values should also be in the "right places", where their contribution to the ILP would be significant (such as critical paths). In this subsection we will present a set of measurements that will indicate that value prediction can have substantial contribution to the exploited ILP.

The gain of ILP that value prediction can accomplish is examined for two different value predictors, the last-value predictor and the stride predictor. Each of these predictors can operate in two modes: the first mode, termed the *scalar generation mode*, allows generation of only a single value prediction for an individual copy of an instruction that resides in the instruction window, while the second mode, termed the *eager generation mode* allows the predictor to generate multiple value predictions assigned to multiple copies of an individual instruction (if any) in the instruction window (e.g. in case of a loop).

Figure 9.1 illustrates the ILP that is gained by employing value prediction when the instruction window size is 40. It also compares the ILP achieved by different predictor schemes (last-value predictor and stride predictor) and prediction modes (scalar mode and eager mode) versus the ILP when value prediction is not employed. Indeed this figure indicates that the potential of value prediction to exceed the current ILP limitations is tremendous, e.g. in the benchmark gcc the ILP is boosted from 14 to nearly 22, in m88ksim from 7 to 34, in perl from 15 to nearly 25 and in vortex from nearly 10 to 33. Figure 9.1 also illustrates that the stride predictor significantly accomplishes better ILP than the last-value predictor in those benchmarks that exhibited spatial value predictability in a form of strides (like m88ksim and vortex) in our previous experiments. For instance, in the benchmark m88ksim the stride predictor boosts the ILP from approximately 7 to 34, while the last-value predictor only gains ILP of 13. This type of spatial value predictability can only be taken advantage of by the stride predictor since the last-value predictor can only exploit temporal value predictability. In the rest of the benchmarks (like go and li) both predictors gain similar ILP with relatively smaller advantage to the stride predictor.

Another major observation shown by these experiments is that the eager mode barely improves the ILP which the last-value predictor gains in all the benchmarks. However, the eager mode significantly improves the ILP which the stride predictor gains in those benchmarks that exhibit spatial value predictability in the form of strides. For instance, in the benchmark m88ksim the stride predictor that operates in eager mode gains ILP of 34 while the same predictor in scalar

mode gains only ILP of 20. This phenomenon seems reasonable, since those instructions with output values exhibiting a tendency to appear in strides are likely to appear recurrently in the instruction window, like instructions in loops, and in order to better exploit them, the predictor should be allowed to operate in eager mode. In the floating point benchmarks, swim and mgrid, all the value predictors gain similar ILP. In the computation phase the floating point operations have less true data dependences and therefore our measurements exhibit more ILP in comparison to the initialization phase that tends to behave like an integer benchmark. It can also be observed that although the stride predictor has significantly exhibited better prediction accuracy than the last-value predictor in ALU instructions, the overall prediction accuracy of both predictors is relatively limited since:

1. Usually the size of basic blocks and loop bodies in floating point programs is relatively big. As a result, the instruction window cannot expose the effect of the loop-carried dependences on the ILP, since it is too small to hold multiple iterations of loop or even several basic blocks
2. Both the last-value predictor and the stride predictor gained relatively limited prediction accuracy in floating-point instructions that may also affect the ILP that they can extract.

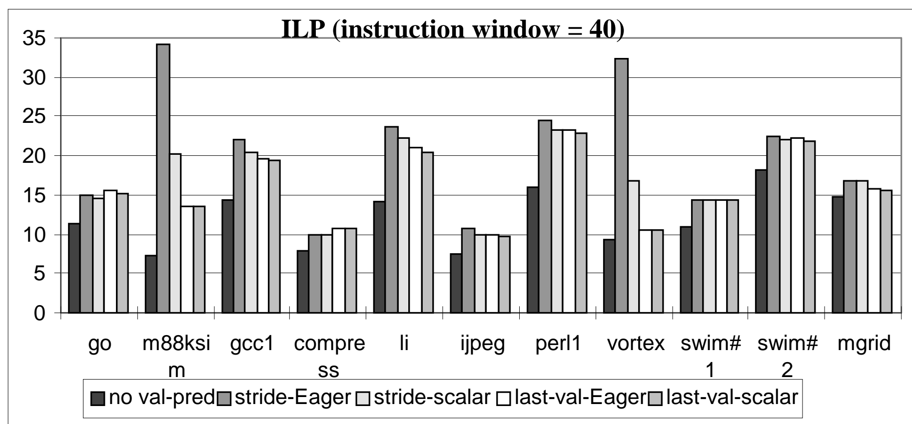

Figure 9.1 - The ILP gained by value prediction when instruction window size is 40.

We have previously indicated that enlarging the instruction window size can enable current processors to look further ahead to find independent candidate instructions for parallel execution. In order to examine how this enlargement affects a machine that employs value prediction, we perform further experiments that are illustrated in figure 9.2. This figure exhibits the same measurements as figure 9.1, however this time the examined machines employ a bigger instruction window with size of 200. These measurements approve that as the instruction window size is increased the extracted ILP grows as well. However, the most interesting observation that these experiments present is that the enlargement of instruction window particularly affects the performance of the eager generation mode and the stride predictor. A bigger instruction window significantly increases the likeliness that it would maintain repeated copies of a same basic block or the same instruction simultaneously, such as multiple iterations of a loop. Such patterns can be usefully exploited by the eager generation mode. This mode allows the predictor to generate multiple value predictions to multiple copies of the same instruction and hence it can better utilize the deeper look-ahead provided by the enlargement of the instruction window. The stride predictor can

also take advantage of these patterns, since appearances of recurrent instructions in the instruction window are also likely to generate output values that progress in strides.

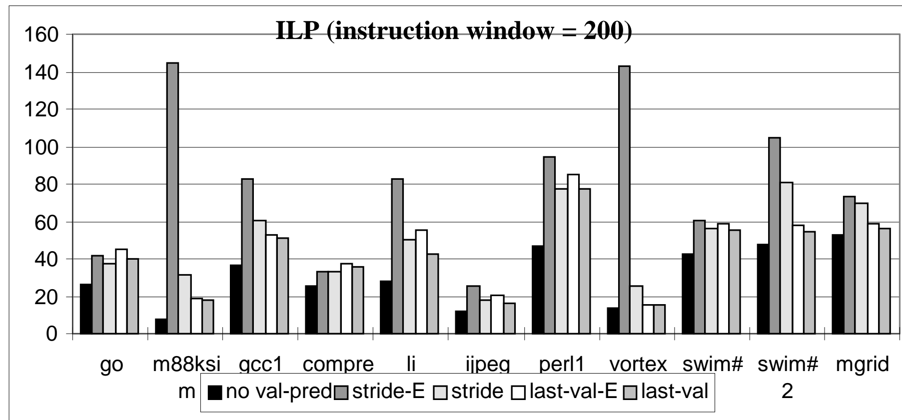

Figure 9.2 - The ILP gained by value prediction when instruction window size is 200.

One can notice that even benchmarks that did not exhibit significant spatial value predictability in a form of strides in the previous experiments, like gcc, li and perl, are significantly affected when they employ both stride predictor and eager generation mode. For instance, in benchmark gcc a stride predictor that operates in eager mode boosts the ILP from 36 to 82, while the same predictor in scalar mode gains ILP of only 60. In addition the gap between the stride predictor and the last-value predictor, that gains ILP of approximately 50, becomes more noticeable. In benchmark li the effect of the eager mode is even noticeable on the last-value predictor. The last-value predictor boosts the ILP of this benchmark from 28 to 55 while the same predictor operating in scalar mode gains only 42. However, the finest ILP among all the predictors in this benchmark is gained by the stride predictor which boosts the ILP to 82. In the benchmarks that exhibited spatial predictability in form of strides (m88ksim and vortex) the effect of the enlargement is the most observable. In the benchmark m88ksim the stride predictor in eager mode boosts the ILP from 7.4 to 144 while the same predictor in scalar mode gains only 31. In the benchmark vortex similar patterns are observed: the ILP is boosted from 13.5 to 142 by the stride predictor operating in eager mode, while the same predictor in a scalar mode gains ILP of nearly 26. In addition, the enlargement of the instruction window affects the extracted ILP in the floating point benchmarks as well. The gap between the stride predictor in eager mode and the other schemes becomes much more significant since the window size enlargement can better expose the loop-carried dependences. The stride predictor boosts the ILP of the benchmarks swim (in the computation phase) from 47 to 104, and in the benchmark mgrid it boosts the ILP from 53 to 73.

The results of the first set of experiments are very encouraging about the potential of value prediction to boost the ILP beyond the dataflow graph limitations. In addition, till now several studies such as [John90] indicated that large instruction windows may not be cost-effective since they do not offer significant boost in the ILP to justify its hardware-cost. When value prediction is employed this claim may no longer be valid. It is true that value prediction may increase the hardware complexity, however it offers tremendous potential to extract ILP beyond the present limitations. In addition, we have seen that both stride predictor and eager generation mode may significantly gain better ILP particularly when the size of the

instruction window is increased. However, in order to maintain these mechanisms the cost in terms of hardware complexity can be expensive. One of the future directions that we consider is to maintain a hybrid approach that consists of both predictor schemes (last-value predictor and stride predictor) and both value prediction generation modes (scalar mode and eager mode). This approach, motivated by our experiments, indicates that on one hand the absolute number of instructions exhibiting spatial value predictability in form of strides is relatively smaller than those exhibiting temporal predictability, however on the other hand value prediction based on strides can significantly boost the ILP particularly in big instruction windows. Hence, in order to take advantage of these observations a machine could partition the limited resources assigned to the value prediction schemes more efficiently, e.g. by employing a small prediction table for the stride-predictor and a bigger table for the last-value predictor and only allowing value predictions based on strides to be generated in eager mode. These issues and many other implementation consideration issues are left for possible future studies.

9. 2. The impact of filtered value prediction

Our previous experiments in section 7 have exhibited that the probability to correctly predict outcome values is not uniformly distributed among the instructions in a program, and in fact all the instructions that generate outcome values can be classified into two major classes: the class of the likely predictable ones and the class of the unlikely predictable instructions. We also indicated that classifying instructions is very important in order to 1. better utilize the limited size of the prediction tables and 2. reduce the number of mispredictions. In order to perform this kind of classification, it was proposed to filter the unpredictable set of instructions by employing filtering mechanism such as the mechanism that was described in section 7 (saturated counter). In this section we will examine the effect of filtering on the ILP that can be extracted. Unlike the previous set of measurements where it was assumed that the value predictor is allowed to handle every instruction that generates value to be written to a register, in these measurements a 2-bit saturated counter is attached to each entry in the value prediction table and upon the present state of the counter it is decided whether to take the prediction or to ignore it (see section 7 for more details).

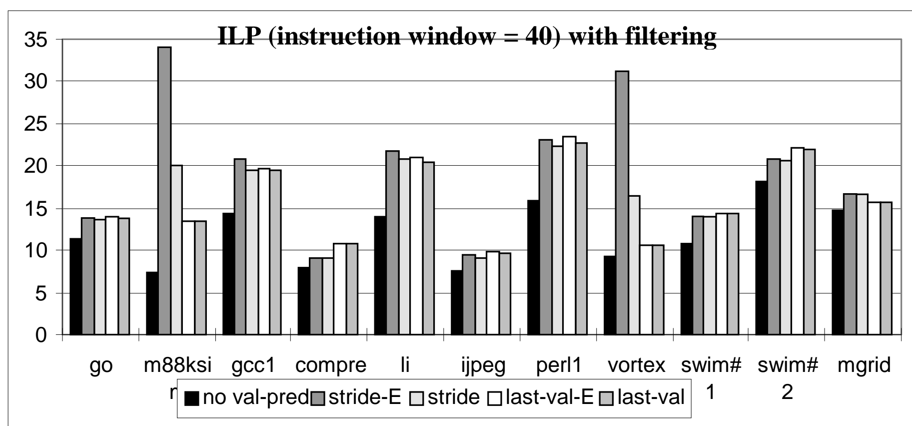

Figure 9.3 - The effect of filtering on the gained ILP.

Figure 9.3 illustrates the new ILP measurements when the instruction window size is 40 and the value predictors employ a set of 2-bit saturated counters for filtering the unpredictable instructions. These results can be compared to the measurements of figure 9.1. As it was expected, the effect of filtering on the ILP exhibited in this figure is barely noticeable. In addition to these measurements, our experiments indicate that such a filtering mechanism can filter out approximately 94% of the value misprediction at the cost of filtering out 5% out of the correct value prediction. These measurements are very encouraging about the potential of filtering to avoid value misprediction. In addition, as a future direction of this work we also intend to examine another methodology for filtering by employing compiler directives based on program profiling in order to assist the processor to classify the likely predictable class of instructions (section 7).

We can conclude the contribution of this section as follows:

- We presented a set of measurements indicating that value prediction has the potential to significantly enlarge the extracted ILP beyond the ILP that "restricted dataflow machines" gain.
- We have introduced two different value prediction modes: the scalar generation mode and the eager generation mode, and examined their impact on the extracted ILP.
- Our measurements indicate that the stride predictor significantly accomplishes better ILP than the last-value predictor in those benchmarks that exhibited spatial value predictability in a form of strides, while in the rest of the benchmarks both predictors gain similar ILP with relatively smaller advantage to the stride predictor.
- We observed that the eager generation mode particularly improves the ILP that the stride predictor gains in those benchmarks which exhibited spatial value predictability in form of strides.
- Enlarging the instruction window, improves significantly the ILP gained by the eager generation mode and the stride predictor. We also observed that programs which exhibit temporal value predictability can gain certain benefit from eager generation mode as well.
- Our measurements indicated that the use of saturated counters as a filtering mechanism can be attractive to avoid value misprediction. This scheme was found to be capable of eliminating a considerable part of the misprediction with a negligible cost on the ILP.

10. Conclusions and research plan

In this work we have presented the new concept of *value prediction* that may lead to new directions in designing future computer architectures. This new concept is associated to a phenomenon which indicates that programs tend to reuse their recently generated values during their execution. By exploiting this phenomenon, we attempt to collapse true-data dependences by performing speculative execution based on predicted values. So far, all modern computer systems were based on the following principles: (1) the dataflow graph of a sequential program forms an upper fundamental bound of the instruction-level parallelism, and (2) in order to guarantee the correct semantics of the program, instruction execution order must not contradict the sequence presented by the dataflow graph. The importance of the *value prediction* concept is that it strikes these fundamental principles and so opens new horizons for future computer architectures.

Throughout this work, we have examined the concept of value prediction from two different perspectives: 1. we have studied the characteristics of the phenomenon from the viewpoint of the program code and 2. we have started to explore how these characteristics can be taken advantage of in order to improve the instruction-level parallelism that superscalar processors extract out of a sequential program.

Although the concept of value prediction was investigated in parallel to us by a group of researches in CMU, our results, as we have shown, are strengthened by their reports, and in addition we provide substantial achievements and contributions beyond theirs.

10. 1. Contributions

The main issues we have pointed out during this work can be summarized as follows:

1. We have better formalized and extended the concept of value prediction and provided a new related terminology:
   - We distinguished between two different kinds of speculative execution: speculative execution based on branch prediction and speculative execution based on value prediction.
   - We introduced the notion of value predictability and distinguished between two different types of value predictability: temporal value predictability and spatial value predictability.
2. We have revealed substantial evidence confirming that programs tend to reuse their recently generated values and to exhibit predictable patterns of value. In addition we have shown that programs can exhibit two kinds of value predictability patterns, temporal and spatial. We also examined the distribution of these properties between different programs, instruction types and value data types.
3. We have introduced various value predictors and examined how they exploit different value predictability properties.
4. We indicated that speculative execution based on value prediction is feasible since it uses an available technology.
5. We have shown that the prediction accuracy does not distribute uniformly among the instructions in a program. Most programs exhibit two sets of instruction, high value-predictable instructions and highly unlikely-predictable ones. These observations motivate us to develop classification mechanisms that could filter out the unlikely predictable instructions from being candidates for value prediction.
6. Our preliminary observation indicates that different input files do not greatly change the value predictability of a program. This observation is encouraging for our future intention to use profiling-based compiler techniques that could classify the value predictability of instructions based on previous runs of the programs.
7. We showed that the use of value prediction techniques have substantial contributions in respect to the ILP that a processor can extract:
   - Our measurements of the average ILP showed a significant improvement.
   - We have presented two different value prediction modes: the scalar generation mode and the eager generation mode, and examined their impact on the extracted ILP.

- Our measurements indicate that the stride predictor significantly accomplishes better ILP than the last-value predictor in those benchmarks that exhibited spatial value predictability in a form of strides, while in the rest of the benchmarks both predictors gain similar ILP with relatively smaller advantage to the stride predictor.
- We observed that the eager mode particularly improves the ILP that the stride predictor gains in those benchmarks which exhibited spatial value predictability in form of strides.
- Enlarging the instruction window, improves significantly the ILP gained by the eager generation mode and the stride predictor. We also observed that programs which exhibit temporal value predictability can gain certain benefit from eager generation mode as well in this case.
- Our measurements indicated that the use of saturated counters as a filtering mechanism can be attractive to avoid value misprediction. This scheme was found to be capable of eliminating a considerable part of the misprediction with a negligible cost on the ILP. These measurements are very encouraging about the potential of filtering to reduce the number of value mispredictions.

10. 2. Research plan

So far we have presented the characteristics of value prediction and provided indications as to how well it can be used to enhance future computer architectures. Now that we have accomplished the establishment of the basis and the evidence of the new phenomenon, we intend to continue our research towards achieving better understanding about the capabilities of value prediction and how it can be employed. Since the concept of value prediction is new, there are a variety of possible directions to be explored. In the following subsection we will introduce some of these possible directions. At this stage of the research we do not want to limit ourselves to any one particular direction, since we want to leave ourselves enough alternatives.

10. 2. 1. Further study on the characteristic patterns of value predictability

In order to examine the potential of each of the alternatives that will be described later in this section, further work should be done on the characterization of value predictability. A possible direction is to examine value predictability patterns in higher levels of programming languages (like C and FORTRAN), instead of focusing on the pattern that appears in the low level machine code (assembler). This can possibly be done by collecting the predictability information of the program in the low-level code and mapping this information from the low-level code to the higher level. Exploring such patterns can provide us with valuable knowledge that can be exploited by the user or the software for writing computer programs.

10. 2. 2. Combining program profiling and compiler support

We suggest examining the potential of combining program profiling and compiler support with value prediction. This integration consists of the following future aspects:

Our experiments revealed an important property about value prediction which indiciates that the prediction accuracy does not distribute uniformly among instructions in a program. We observed that some operations produced outputs which are highly predictable, while other operations produced data which we could not predict. Thus, we were motivated to develop classification mechanisms that could: (1) reduce (filter) value mispredictions, (2) better utilize the prediction table, and (3) increase the effective prediction accuracy. A possible scheme that was examined was staurated counters. As part of our future research, we consider a different type of classification technique. The proposed technique is based on program profiling that will collect information about the tendency of instructions to be predictable according to previous runs of the program. This information will be passed to the processor, for example, through special directives that will be inserted into the opcode of instructions by the compiler, and provide hints as to whether an instruction exhibits value predictability based on stride patterns, or reuse of the recent value. In order to explore the proposed technique we intend to accomplish the following steps:

1. Collect information about the value predictability of instructions under different runs of a program with different input files and parameters.
2. Examine the correlation between the collected information under different runs.
3. Develop mechanisms that can utilized this information at run time.

We expect to obtain valuable information about the capability of program profiling from the first two steps. Our current observation supports our approach for using profiling since it indicated that different input files do not greatly change the value predictability. The mechanisms and techniques we will choose for the third step depends upon the quality of the profiling.

We consider exploiting further the use of the program profiling technique in order to develop better instruction scheduling algorithms. Current instruction scheduling algorithms attempt to optimize the execution of the critical path in the dataflow graph and efficiently share the machine resources between the critical path and the other paths in the graph. When we integrate the information about the value predictability of instructions to the instruction scheduler of the compiler (assuming that this information was collected by profiling), we may observe in several circumstances that the conventional critical path may not be the longest execution path anymore (since it can consist of true-data dependent instructions that may be executed in parallel). In addition, when instructions are not necessarily executed according to the sequence of the dataflow graph, it cannot be ensured that the conventional instruction scheduler shares the machine resources efficiently between the different paths. Therefore, the compiler's accessibility to the value predictability information calls to revise: (1) the criteria to determine which path in the dataflow graph is the critical path, and (2) the instruction scheduling algorithm itself.

10. 2. 3. Developing hybrid predictors and value generation modes

An alternative or additional proposed direction that will be examined is the developing hybrid vale predictor and hybrid value prediction generation modes. Our studies of the stride patterns indicate that nearly 15% of the correct value predictions were based on non-zero strides while the rest were zero strides. Due to this observation we consider developing hybrid value predictors that combine both the stride predictor and the last-value predictor. This combination will lead us to

use the prediction table resources more efficiently. In addition to the hybrid predictor we will consider examining a hybrid approach for value prediction generation modes (scalar mode and eager mode). This approach is also strengthened by our experiments that indicate that on one hand the absolute number of instructions that exhibit spatial value predictability in the form of strides is relatively smaller than those that exhibit temporal predictability, however on the other hand value prediction based on strides can significantly boost the ILP particularly in big instruction windows.

10. 2. 4. Microarchitecture aspects

We believe that in order to better utilize the capabilities of value prediction different microarchitecture aspects should be examined. Some of these aspects are introduces as follows:

1. What should the status be in the instruction window (reservation station) of instructions that are executed speculatively based on value prediction? - Should they evacuate the instruction window or continue to occupy an entries till they are validated?
2. Is it possible to overcome or avoid the value misprediction penalty? - Should the processor wait until the predicted value is truly computed and only afterwards validate it, or perhaps it may attain an indication about the correctness of the prediction during the true computation of the predicted value?
3. In what manner should the predicted values be validated? - We can perform the validation procedure of several predicted values in either a serial or parallel manner. In addition, in several cases there can be links between the predicted values which dictate the manner of the validation.

We are convinced that exploring these aspects can provide a significant contribution in order to better exploit the concept of value prediction.

10. 2. 5. Analytical model

So far no attempt has been made to develop an analytical model that could predict the benefit of using value prediction. We hope to develop such a model, so that the value of some fundamental parameters can be examined analytically.

10. 2. 6. Combining trace cache with value prediction for future processor architecture

We have shown how value prediction can exceed one of the fundamental limitations on instruction-level parallelism. Current processor architectures face another limitation that may reduce their performance particularly when value prediction is employed. We have indicated in Section 2 that superscalar processors seek to execute a serial program in the sequence exhibited by the dataflow graph, however since the sequence in which instructions are stored in memory may not necessarily correspond to the dataflow graph sequence, the instruction fetch may become a bottleneck in the system. We are convinced that this limitation will be become much more severe when the processor employs value prediction. Value prediction will attempt to increase the instruction execution rate and eventually may significantly increase the instruction consumption rate from the instruction fetcher. A recent work ([Rote96]) proposed a certain mechanism, termed *trace cache*, that can improve the effective production rate of the instruction fetcher. However this work did not take into account the significance of this

mechanism when value prediction is employed and how it can be integrated with value prediction mechanisms. A possible direction that we can navigate our future work is find answers to these open questions.

References

[Adam74] T. L. Adam, K. M. Chandy and J. R. Dickson. A Comparison of List Scheduling for Parallel Processing Systems. Communications of ACM, vol. 17, December, 1974, pp. 685-690.

[Aho86] A. V. Aho, R. Sethi and J. D. Ullman. Compilers: Principles, Techniques and Tools. Reading, MA: Addison-Wesly, 1986.

[Aike88] A. Aiken and A. Nicolau. Perfect Pipelining: A New Loop Parallelization Technique. In H. Ganzinger (ed.) Proceedings of the 2nd European Symposium on Programming. pp. 221-235. New-York: Springer - Verlag, March, 1988.

[And95] H. Ando, C. Nakanishi, T. Hara and M. Nakaya. Unconstrained Speculative Execution with Predicated State Buffering. In proceeding of the 22nd International Symposium on Computer Architecture. June, 1995, pp.126-137.

[Blai94] R. J. Blainey. Instruction Scheduling in the TOBEY Compiler. IBM J. Res. Develop., Vol. 38, No. 5, Spetember, 1994, pp. 577-593.

[Chai81] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins and P. W. Markstein. Register Allocation via Coloring. Computer Languages, Vol. 6, no. 1, 1981, pp. 47-57.

[Chan91] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter and W. W. Hwu. IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors. Proceedings of the 18th International Symposiu on Computer Architecture, pp. 266-275, May, 1991.

[Chan96] P. -Y. Chang, M. Evers and Y. N. Patt. Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference. The Conference on Parallel Architectures and Compilation Techniques (PACT), 1996, pp.48 - 57.

[Char81] A. E. Charlesworth. An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS Family. Computer, Vol. 14, September, 1981, pp.18-27.

[Chen95] T. F. Chen and J. L. Bear. Effective Hardware-based Data Prefetching for High-performance Processors. IEEE Transactions on Computers, 44(5): 609-623, May, 1995.

[Chow90] F. C. Chow and J. L. Hennessey. The Priority-Based Coloring Approach to Register Allocation. ACM Transactions on Programming Languages and Systems. Vol. 12, No.4, October, 1990, pp. 501-536.

[Coff76] E. J. Coffman, editor. Computer and Job-shop Scheduling Theory. John Wiley and Sons, New-York, NY, 1976.

[Davi81] S. Davidson, D. Landskov, B. D. Shriver and P. W. Mallet. Some Experiments in Local Microcode Compaction for Horizontal Machines. IEEE Transactions on Computers, Vol. C-30, no. 7, July, 1981, pp. 460-477.

[Diep95] T. A. Diep, C. Nelson and J. P. Shen. Performance Evaluation of the PowerPC 620 micro-architecture. In proceeding of the 22nd International Symposium on Computer Architecture. June, 1995, pp. 163-174.

[Ditz87] D. R. Ditzel and H. R. McLellan. Branch Folding in the CRISP microprocessor: Reducing the Branch Delay to Zero. Proceeding of the 14th International Symposium on Computer Architecture. June, 1987.

[Edmo95] J. H. Edmondson, P. Rubinfeld, R. Preston and V. Rajagopalan. Superscalar Instruction Execution in the 21164 Alpha Microprocessor. IEEE Micro, April, 1995, pp. 33-43.

[Elli86] J. R. Ellis. Bulldog: A Compiler for VLIW Architecture. MIT Press, Cambridge, Mass., 1986.

[Fish79] J. A. Fisher. The Optimum of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling with Resources. Ph.D. dissertation, Technical Report COO-3077-161. Courant Mathematics and Computing Laboratory, New York University, New York, October, 1979.

[Fish92] J. A. Fisher and S. M. Freudenberger. Predicting Conditional Branches from Previous Runs of a Program. Proceeding of the 5th Conference of Architectural Support for Programming Languages and Operating Systems. IEEE/ACM, Boston, October 1992.

[Gabb96] F. Gabbay and A. Mendelson. A system and Method for Concurrent Processing. Submitted to patent registration.

[Good88] J. R. Goodman and W.-C. Hsu. Code Scheduling and Register Allocation in Large Basic Blocks. Proceeding of the 1988 International Conference on Supercomputing. July, 1988, pp. 442-452.

[Gwen95] L. Gwennap. Intel's P6 Uses Decoupled Superscalar Design. Microprocessors Report vol. 9, num. 2, February 16, 1995.

[Harb82] S. Harbison. An architectural Alternative to Optimizing Compilers. Proceeding of the International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 57-65, March, 1982.

[Henn90] John L. Hennessy and David A. Patterson. Computer Architect a Quantitative Approach. Morgan Kaufnab Publishers, Inc., 1990.

[Henn95] John L. Hennessy and David A. Patterson. Computer Architect a Quantitative Approach. Morgan Kaufnab Publishers, Inc., Second edition, 1995.

[Hsu86] P. Y. T. Hsu and E. S. Davidson. Highly Concurrent Scalar Processors. Proceedings of the 13th International Symposium on Computer Architecture, pp. 386-395, June, 1986.

[Hwu88] W. Hwu, and P. P. Chang. Exploiting Microprocessor Microarchitectures with a Compiler Code Generator. Proceeding of the 15th International Symposium on Computer Architecture. June, 1988, pp. 45-53.

[John90] Mike Johnson. Superscalar Microprocessor Design. Prentice Hall, Englewood Cliffs, 1990, N.J.

[Joup89] N. P. Jouppi and D. W. Wall. Available Instruction-Level Parallelism for Superscalar and Superpipelines Processors. Proceeding of the 3rd Conference of Architectural Support for Programming Languages and Operating Systems. IEEE/ACM, April, 1989, pp. 272-282.

[Kell75] R. M. Keller. Look-ahead Processors. Computing Surveys, volume 7, no.4, December, 1975, pp. 177-195.

[Lam88] M. S. Lam. Software pipelining: An effective scheduling technique for VLIW processors. SIGPLAN Conference on Programming Languages Design and Implementation, ACM (June), Atlanta, Ga., 318-328.

[Lam89] M. Lam. A Systolic Array Optimizing Compiler. Boston: Kluwer, 1989.

[Lam92] M. S. Lam and R. P. Wilson. Limits of Control Flow on Parallelism. Proceeding of the 19th International Symposium on Computer Architecture. May, 1992. pp. 46-57.

[Lipa96a] Mikko H. Lipasti, Christopher B. Wilkerson and John Paul Shen. Value locality and load value prediction. In proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS- VII), October 1996.

[Lipa96B] Mikko H. Lipasti, Christopher B. Wilkerson and John Paul Shen. Exceeding the dataflow limit via value prediction. In proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, December 1996.

[Mahl92] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau and M. S. Schlansker. Sentinel Scheduling for VLIW and Superscalar Processors. Proceeding of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 238-247, October, 1992

[Mahl95] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August and W. W. Hwu. A Comparison of Full and Partial Predicated Execution Support for ILP Processors. In proceeding of the 22nd International Symposium on Computer Architecture. June, 1995, pp. 138-149.

[McFa86] S. McFarling and J. Hennessy. Reducing the Cost of Branches. Proceedings of the 13th International Symposium on Computer Architecture. June, 1986. pp. 396-403.

[Mend96] A. Mendelson and F. Gabbay. Speculative Execution based on Value Prediction. Technion EE Department Technical Report.

[Moud94] M. Moudgill. Implementing and Exploiting Static Speculation on Multiple Instruction Issue Processors. Ph.D. Dissertation, University of Cornell Technical Report.

[Nico84] A. Nicolau and J. A. Fisher. Measuring the Parallelism Available for Very Long Instruction Word Architectures. IEEE Transactions on Computer, vol. C-33, November, 1984, pp. 968-976.

[Pint96] S. S. Pinter and A. Yoaz. Tango: a Hardware-based Data Prefetching Technique for Super-scalar Processors. In proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, December 1996.

[Rau81] B. R. Rau and C. D. Glaeser. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. Proceeding of the 14th Annual Workshop on Microprogramming, October, 1981, pp. 183-198.

[Rise72] E. M. Riseman and C. C. Foster. The inhibition of Potential Parallelism by Conditional Jumps. IEEE Transactions on Computer, vol. C-21, December, 1972, pp. 1405-1411.

[Rote96] E. Rotenberg, S. Bennett, J. Smith. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. In proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, December 1996.

[Simo95] M. Simone, A. Essen, A. Ike, A. Krishnamoorthy, T. Maruyama, N. Patkar, M. Ramaswami, M. Shebanow, V. Thirumalaiswamy and D. Tovey. Implementation Trade-offs in Using Restricted Data Flow Architecture in High Performance RISC Microprocessor. In proceeding of the 22nd International Symposium on Computer Architecture, June 1995, pp. 151-162.

[SHADE] Introduction to Shade, Sun Microsystems Laboratories, Inc. TR 415-960-1300, Revision A of 1/Apr/92.

[Smit84] A. Smith and J. Lee. Branch Prediction Strategies and Branch-Target Buffer Design. Computer 17:1, January 1984. pp. 6-22.

[Smit81] J. E. Smith. A Study of Branch Prediction Techniques. In proceeding of the 8th International Symposium on Computer Architecture, June 1981.

[Smit89] M. D. Smith, M. Johnson and M. A. Horowitz. Limits on Multiple Instruction Issue. Proceeding of the 3rd International Conference on Architectural Support for Programming Languages and Operating System. April, 1989, pp. 290-302.

[Sohi90] G. S. Sohi. Instruction Issue Logic for High-Performance, interruptible, multiple functional unit, pipelined computers. IEEE Transactions on Computers 39:3, March, 1990, pp. 349-359.

[Tjad70] G. S. Tjaden and M. J. Flynn. Detection and Parallel Execution of Independent Instructions. IEEE Transactions on Computer, C-19:10, October, 1970, pp. 889-895.

[Toma67] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. IBM J. Research and Development 11:1, pp. 25-33, January, 1967.

[Wall91] D. W. Wall. Limits of Instruction-Level Parallelism. Proceedings of the 4th Conference on Architectural Support for Programming Languages and Operating Systems. April, 1991. pp. 248-259.

[Weis87] S. Weiss and J. E. Smith. A Study of Scalar Compilation Techniques for Pipelined Supercomputers. Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems. October, 1987, pp. 105-109.

[Yeh91] T. Y. Yeh and Y. N. Patt. Two-Level Adaptive Training Branch Prediction. In proceedings of the 24th Annual ACM/IEEE International Symposium on Microarchitecture, December 1992.

[Yeh92] T. Y. Yeh and Y. N. Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. Proceedings of the 19th International Symposium on Computer Architecture. May, 1992. pp. 124-134.

[Yeh93] T. Y. Yeh and Y. N. Patt. A Comparison of Dynamic Branch Predictors that Uses Two Levels of Branch History. Proceedings of the 20th International Symposium on Computer Architecture. May, 1993. pp. 257-266.