The Performance Potential of Data Dependence Speculation & Collapsing

Yiannakis Sazeides
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
1415 Engr. Dr.
Madison, WI 53706
yanos@ece.wisc.edu

Stamatis Vassiliadis
Department of Electrical Engineering
Delft University of Technology
Mekelweg 4, 2628 CD Delft
The Netherlands
stamatis@duteca.et.tudelft.nl

James E. Smith
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
1415 Engr. Dr.
Madison, WI 53706
jes@ece.wisc.edu

The Performance Potential of Data Dependence Speculation & Collapsing

Yiannakis Sazeides
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
1415 Engr. Dr.
Madison, WI 53706
yanos@ece.wisc.edu

Stamatis Vassiliadis
Department of Electrical Engineering
Delft University of Technology
Mekelweg 4, 2628 CD Delft
The Netherlands
stamatis@duteca.et.tudelft.nl

James E. Smith
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
1415 Engr. Dr.
Madison, WI 53706
jes@ece.wisc.edu

Abstract

Two hardware methods for remedying the effects of true data dependences are studied. The first method, dependence speculation, is used to eliminate address generation-load dependences. This is enabled by address prediction that permits load instructions to proceed speculatively without waiting for their address operands. The second technique, dependence collapsing, is used to eliminate data dependences by combining a dependence among multiple instructions into one instruction. The potential of these techniques for improving processor performance is demonstrated via trace-driven simulation. When both techniques are used with maximum issue widths of 4, 8, 16, and 32, the overall speedups in comparison to a base instruction level parallel machine are 1.20, 1.35, 1.51, and 1.66, respectively. In general, dependence collapsing contributes the majority of the improvement in performance. Under the dependence collapsing model, 29% to 47% of the total number of instructions in a trace may be collapsed. The distance separating the collapsed instructions is nearly always less than 8. Our experimentation also suggests that further performance improvements can be achieved by incorporating mechanisms that increase the address prediction rate.

1. Introduction

An execution of a computer program defines a dynamic dataflow or dependence graph, that reflects the true data and control dependences. That is, the dynamic dependence graph contains all the dynamic instructions as nodes, with an arc from one instruction to another if there is a true data or control dependence. If we label an arc with a time equal to the execution time of the instruction producing the result, then, in theory, the minimum execution time of the program is the length of the longest path (i.e. the "critical path") through the dependence graph.

Driving towards the critical path execution time requires discovery of parallelism among instructions. Consequently, many techniques for instruction level parallelism (ILP) are aimed at discovering and executing parallel instructions, both in hardware and software – e.g. multiple reservations stations, issuing multiple instructions out of order, software pipelining, loop unrolling, etc. A different approach is the use of techniques that improve performance by essentially "restructuring" the dependence graph. There are two basic ways hardware can be used to do this:

- The first method effectively removes a dependence arc by predicting the value carrying the dependence and proceeding with speculative execution. If the prediction is correct, then the critical path is decreased – possibly below the theoretical minimum (at least locally). If the prediction is incorrect, then recovery is imperative and the critical path is not decreased; execution time can in fact be increased in a real implementation by using resources needlessly. This technique has been used for branches (control dependences) for some time, but has also been proposed for memory addresses [5] and for data loaded from memory [9].

- The second method can reduce the latency by combining a dependence among multiple dependent instructions into a single instruction. Functionality for the collapsing of multiple operations has been advanced in a number of proposals for floating point and fixed point instructions [14, 16, 19].

The performance limit studies published in the literature [1, 3, 6, 7, 20] are primarily based on the assumption that true dependence paths through the dynamic dependence graph place a limit on performance. In addition to theoretical limits under ideal assumptions, limit studies also explore the envelope of practical performance by placing some constraints on hardware execution resources, for example by limiting the "window" of instructions that may simultaneously be considered for parallel execution. Significant performance is achieved with perfect branch prediction, but gains are diminished when using realistic prediction. Furthermore, though predicting branches removes some control dependences, data dependences remain as an impediment for increased performance.

In this work, we investigate the potential performance improvements that can be achieved by the combined use of two hardware approaches that restructure the dynamic dependence graph: data dependence speculation (*d-speculation*) and data dependence collapsing (*d-collapsing*). To illustrate the two approaches, consider the code segment and its dynamic dependence graph shown in Figures 1.a-b. The d-speculation approach can be used to predict the addresses for memory instructions, thus enabling the removal of the arc(s) between dependent address generation and load instruction (in Figure 1.c this results in the removal of the arc between instructions 1 and 2). D-speculation can also be used to predict data values such as those loaded from memory (in Figure 1.d this results in the removal of the arc between instructions 2 and 3) and in general the data result of any instruction. The d-collapsing approach can be used to combine the dependence of multiple instructions. In the above example the dependence between instructions 3 and 4 can be collapsed by executing *4'. [8+Ra+Rd] = Re* instead of *4. [8+Rc] = Re*. The effect on the dependence graph is shown in Figure 1.e. Note that the dependence to instruction 4' comes from node 2 instead of node 3.

It is worthwhile to observe that it is sometimes possible to eliminate nodes in a dynamic dependence graph. For instance, with the collapsing of the dependence between instructions 3 and 4, if the result of instruction 3 is not needed elsewhere then 3 need not be executed. This restructuring is shown in Figure 1.f. Instructions whose execution is necessary to verify a prediction always need to be executed (such as instruction 1 in 1.c).

The remainder of the paper is organized into the following sections. Section 2 reviews related work. In Section 3, we discuss the speculation and collapsing mechanisms studied in this paper. Section 4 addresses the experimental framework and assumptions. Section 5 contains the discussion of the obtained results. Section 6 will conclude the study.

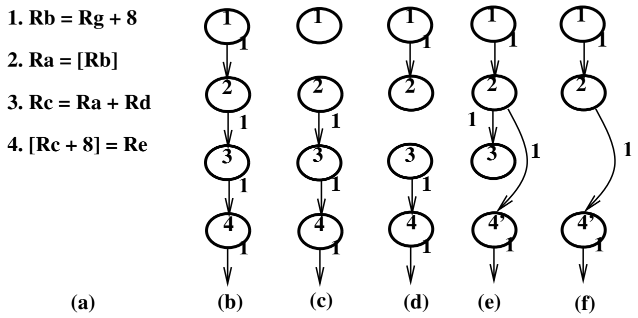

Figure 1. (a) Sample Dynamic Instruction Stream (b) Dynamic Dependence Graph (c) D-Speculation Predict Address (d) D-Speculation Predict Value to be Loaded (e) D-Collapsing (f) Node Elimination

2. Related Work

Several schemes for reducing the effects of execution and memory dependences have been proposed [2, 5, 9, 19]. These techniques are claimed to have minimal or no impact on the clock cycle. Mechanisms for remedying the effects of memory dependences using early memory loads were introduced in [2, 5]. In [5] the combination of three strategies is proposed. The first identifies data dependences required for address generation that can be resolved immediately. The second is used for the case in which dependences can not be resolved immediately; it predicts the address using a table where memory access patterns are stored. The third strategy removes load operations from the instruction stream in advance (possibly multiple cycles before the actual issue of loads). This approach allows the processor to execute load instructions in parallel with instructions preceding the loads, achieving a zero cycle load execution time when the speculation is found to be correct. Austin [2] exploits the fact that small offsets are suitable for simple fast address calculation and determines memory addresses early in the pipeline for those cases. This enables the load instructions to issue early and effectively hide the load-use latency. A different approach that predicts data values instead of addresses has been proposed in [9] for invariant data values loaded from memory. Numerous other schemes for prefetching data in the memory hierarchy have been proposed – for further references see [4, 12]. Prefetching schemes do not consider the possibility of speculative instruction execution using predicted addresses. The address prediction techniques used for prefetching can be applicable for prediction in d-speculation, however.

Data dependence collapsing functionality has been advanced in a number of proposals. Collapsing for specific instances of floating point operations with new instruction definitions has been proposed and implemented in a number of processors. The multiply-add operation was used in the RS/6000 [14, 16] and since then it has been incorporated in other superscalar pipelined processors [21]. A general scheme capable of collapsing a dependence pair involving fixed point arithmetic and logical instructions was proposed in [19]. The proposed solution takes into account a general CISC instruction set that includes the functionality of RISC and post-RISC based instruction sets. The performance of such a proposal for some realistic machine organizations was reported in [10] and a single cycle implementation of a subset of this proposal was implemented in POWER 2 [21]. A device that includes the majority of the functions a collapsing fixed point unit can perform, was proposed in [17].

Several studies have examined the limits of instruction level parallelism with a variety of control and resource models [1, 3, 6, 7, 20]. However, no limit study has examined the impact of data dependence collapsing and speculation on instruction parallelism. A related work [15] used static program representation and profiling weights to calculate the frequencies of sequences (pairs, triples and quads) of dependent instructions within and across basic blocks. This work however did not examine the impact on latency/parallelism. In [10, 18], performance studies were conducted to characterize the effects of interlock (dependence) collapsing functionality on parallelism for a variety of resource and control models. The dependence collapsing performed in these studies was restricted to collapsing only consecutive instructions within a single basic block.

The models used in the work presented here differentiate from previous studies of data speculation and collapsing in the following ways:

- **D-Speculation:**
  - Load speculation is performed selectively based on a confidence indication.
- **D-Collapsing:**
  - Non-consecutive instructions can be collapsed.
  - Dependence collapsing may be performed across basic block boundaries.
  - Shift operations can be collapsed in addition to fixed point arithmetic and logic operations.
  - Three dependent instructions may be collapsed, in addition to two dependent instructions as in previous work.
  - Zero operands are detected and used to enable further collapsing.
- **Combining speculation and collapsing:**
  - We study the combination and interaction of d-speculation and d-collapsing techniques.

3. Speculation and Collapsing Mechanisms

In this section we provide some information regarding the specific mechanisms as they are modeled in the simulator used to collect the results.

D-Speculation Mechanisms: Data speculation can be used to predict data and address values. In this study one mechanism, called **load-speculation**, is directed at memory dependences. The aim of load-speculation is to reduce the effect of the latency of *load-use* dependences. This is achieved by predicting addresses and speculatively issuing load operations.

A realistic method for address prediction uses a table for maintaining information about strides used for memory accesses. The table we consider is a 4096 entry direct mapped table. The 14 least significant bits of a load instruction address is the index into the table. The algorithm used for prediction and the table entry information is identical to the *two delta strategy* in [5][1]. The delta size in our table is 32 bits. To the previously proposed table, we add a 2 bit saturating counter for each table entry to represent the confidence in issuing a load with a predicted address. The counter in each table entry is initialized to zero and incremented by 1 (decremented by 2) on a correct (wrong) address prediction. The predicted address is used only when the counter value is greater than 1. This means that as long as the counter indicates low confidence in a prediction, predicted addresses are not used for speculative load issue.

A load calculating its address early enough that it does not require a predicted address is called a *ready* load. All loads update the table state but only not ready loads (i.e. loads that take advantage of speculation) use the table. Therefore, a not ready load will either speculatively issue using the predicted address from the table (if the confidence table value is greater than 1) or wait to issue until its address can be calculated (if the confidence table value is less than or equal to 1). Speculatively issued loads can be divided further into two categories: those that were *predicted correctly* and those *predicted incorrectly*. Subsequent instructions that depend on a speculative load can issue as soon as the data are available from the memory without having to wait for the determination of the correct address. When the correct address becomes available and a misprediction

[1] Note that the load speculation mechanism uses a subset of the mechanisms presented in [5].

is detected, only the instructions dependent on the load are affected. In the simulator, these instructions can not issue before the cycle that the load with the correct address completes execution. No restriction on the number of loads that can be load-speculated is imposed. We report the distribution of loads sorted into the four categories: ready, not predicted, predicted correctly, predicted incorrectly.

For the purpose of comparison with an ideal case, we also simulated the case where all load addresses are predicted correctly (*ideal load-speculation*).

D-Collapsing Mechanisms: Computational data dependences are resolved using the d-collapsing approach. The collapsing of data dependences, theoretically, can be applied to any sequence of dependent instructions. In this study we enabled only the collapsing of dependences between pairs and triples of instructions (in some cases as explained below four dependent instructions can also be collapsed). A dependent sequence of instructions can be described by a n-1 (n to 1) dependence expression where n is the number of operands in the expression. For instance in the following sequence:

```
1.  Rb =    Rd << Rh
2.  Rg =    Rb +  Re
3.  Ra =    Rf -  Rg
```

the dependence expression between instructions 1 and 2 is a 3-1:

```
Rg = (Rd << Rh) + Re
```

and the dependence expression between instructions 1, 2 and 3 a 4-1:

```
Ra = Rf - ((Rd << Rh) + Re).
```

As a result of the collapsing assumed in this paper, the above serial sequence is transformed to the following with all dependences eliminated.

```
1.  Rb = Rd << Rh
2.  Rg = (Rd << Rh) + Re
3.  Ra = Rf - ((Rd << Rh) + Re)
```

It should be noted that it is possible for a dependence between two or three instructions to result in a dependence expression greater than 3-1 or 4-1. A dependence between a pair of instructions can result to a 4-1 dependence, for example, if the instructions are Rb = Ra + Rd and Rc = Rb + Rb, then the computation of Rb requires (Ra + Rd) + (Ra + Rd) which is a 4-1 dependence. A dependence between three operations can result in up to an 8-1 dependence expression. In this study we assume the performance of a collapsing devices for 3-1 and 4-1 dependence expressions that involve the following operation types: shift, arithmetic (not multiply or divide), logical, move, address generation (for loads and stores), and condition code generation for branch instructions. This is a relatively optimistic approach intended to explore the envelope of the performance potential.

The assumed mechanisms extend proposed devices [17] with the ability of collapsing shift operations and 4-1 dependence expressions. Shift operations are added in our study because they appear frequently in the instruction mix of programs (about 6%) and shift distances are dominated by a few values. We believe the potential exists for collapsing shifts with ALU operations – at least for the commonly occurring shift distances. We also consider collapsing performed between consecutive and non-consecutive instructions, within and across basic block boundaries. Furthermore, zero operands are detected in loads and stores operations and reduce the size of the dependence expressions. For instance, in the fragment code below, the dependence expression for instruction 4 is a 5-1 which is not collapsible with the assumed functionality. However, the detection of the zero reduces the expression to a collapsible 4-1 expression. In general the detection of zero operands can reduce the expression size and hence the complexity of the device required to resolve a dependence.

```
1.  Rf =    Rg or 0x288
2.  Rh =    Ra -  1
3.  Rd =    Rf >> Rh
4.  Ra =    [Rd + 0]
```

We divide the dependences collapsed in three broad categories: *3-1*, *4-1* and zero operand detection (*0-op*). We report the contribution for each collapsing category as well as the distance between collapsed instructions.

4. Simulation Framework

To measure the impact of data speculation and dependence collapsing on parallelism we developed a trace driven simulator for the SPARC v.8 Architecture [13]. Our test set includes the benchmarks shown in Table 1. The benchmarks 026.compress, 008.espresso, 0.23 eqntott and 022.li are from the SPECINT92 suite and 099.go and 132.ijpeg are from the SPECINT95 suite. The benchmarks were compiled using the gcc version 2.6.3 at -O4 optimization level (go and ijpeg benchmarks compiled at -O3 optimization). The traces were generated by *qpt2* [8] and do not include system code. For those benchmarks longer than 250 million instructions, only the first 250 million instructions of each benchmark trace were simulated due to time constraints. Nop operations were ignored and are not included in the simulations.

Simulation Methodology: Our simulation methodology is similar to that used by Wall[20]. Instructions are fetched and placed in a "window", from which instructions are chosen to issue. The maximum window size is fixed, and in-

Name	Input File	Flags	Trace Size (Millions)
026.compress	in		88
008.espresso	bca.in		250
023.eqntott	int_pri_3.eqn	-s -.ioplte	250
022.li	li-input.lsp (7 queens)		207
099.go		9 9	122
132.ijpeg	vigo.ppm		250

Table 1. Benchmark Characteristics

Name	Conditional Branches (%)	Predicted Correctly (%)
026.compress	13.2	89.7
008.espresso	18.5	94.1
023.eqntott	27.5	96.0
022.li	15.8	96.8
099.go	13.5	83.7
132.ijpeg	8.97	92.8

Table 2. Benchmark Branch Characteristics

structions are placed into the window to replace instructions as they issue – i.e. the window is kept full. The latency of the different operations is 1 cycle with the following exceptions: loads and multiplications require 2 cycles and divides require 12 cycles.

Simulation Parameters: The simulation model contains a number of parameters that characterize important processor characteristics. These parameters include the maximum number of instructions allowed in the scheduling window, the maximum issue width and the types of data speculation and collapsing schemes used. The maximum issue width establishes the maximum attainable instruction level parallelism. For all configurations simulated, conditional branches are predicted using the bimodalN/gshareN+1 scheme proposed in [11] with 8kByte cost. All other branches and jumps (indirect, unconditional, call and return) are assumed to be always predicted correctly. The performance of the predictor is shown in Table 2. We show the percentage of branch instructions in each trace and the percent of the branches that were predicted correctly.

We assume zero cycle penalty for fetching instructions from the path of correctly predicted branches. In the case of misprediction, instructions following a branch can not issue before or during the cycle the branch instruction issues.

All simulated configurations use ideal register renaming and perfect memory disambiguation. No resource limit was set for the different operation types.

Real load-speculation uses the address prediction table described in the previous section. *Ideal* load-speculation is assumed to speculate correctly every load instruction. Load-speculation in the context of memory disambiguation is modeled as follows. With no load-speculation a load can issue as soon as all its dependences are satisfied; a load-speculated load needs to respect all dependences with the exception of address generation dependences. In the case of address misprediction, or no prediction, a load will issue as if no load-speculation is employed. A speculatively issued load instruction may require resources for both speculation and verification. In this study we account for the resources during the verification phase only.

The simulations were conducted for a base and four different data speculation and dependence collapsing configurations. The different configurations simulated are outlined below. The uppercase letter in parentheses will be used to identify the configurations in graphs to follow.

- base, *(A),*
- base + real load-speculation, *(B),*
- base + d-collapsing, *(C),*
- base + d-collapsing + real load-speculation *(D)*
- base + d-collapsing + ideal load-speculation *(E)*

For each of the above configurations simulations were performed for instruction issue widths of 4, 8, 16, 32 and 2048 (2k). In each case the window size was set to twice the issue width.

5. Simulation Results

This section contains results of our performance simulations. We present results both in terms of instructions per cycle (IPC) and speedup versus the base superscalar machine. The speedups are with respect to the base configuration with same issue width. For both measures, we summarize results by taking the harmonic mean over the benchmark set.

5.1. Overall Performance

For the five machine configurations studied (A through E), Figures 2 and 3 show the harmonic mean instructions per cycle and speedups versus the base machine. Results for maximum issue widths 4, 8, 16, 32, and 2K per cycle are plotted. The results suggest the following:

- The maximum potential of d-speculation and d-collapsing (point E) is quite high, with speedups from 1.25 (issue width 4) to 2.95 (issue width 2K). Note that these are compared with base superscalar implementations that are quite optimistic – with the major (non-ideal) limitation being realistic branch prediction.
- If a realistic load-speculation mechanism is used with d-collapsing (point D), speedups are still significant (1.2–1.9).

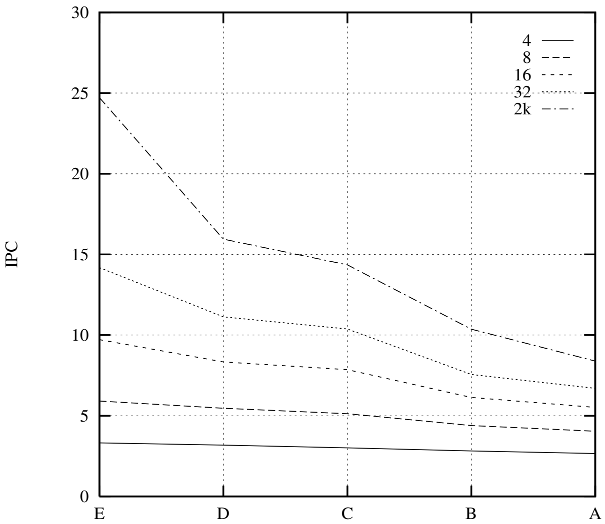

Figure 2. IPC for the Different Configurations and Issue Widths

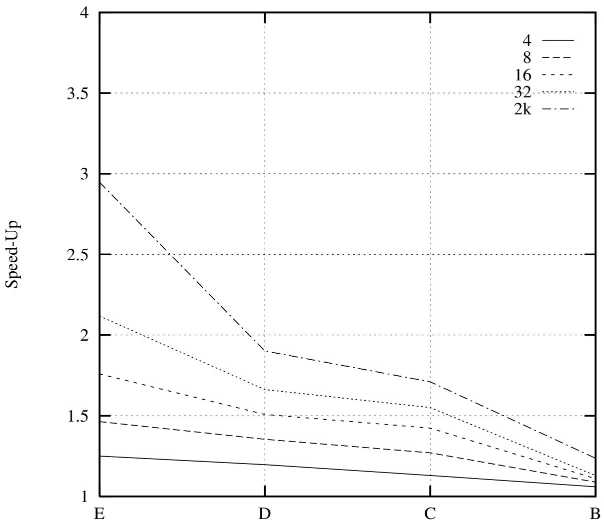

Figure 3. SpeedUp over the Superscalar Base Machine (A)

- The combined use of d-collapsing and d-speculation provide speedups roughly equal to the sum of the individual speedups of d-collapsing and d-speculation (compare point D and the sum of points B and C in Figure 3).

Although the use of realistic load-speculation and d-collapsing provide significant speedup, as pointed out above, there is also significant performance drop when going from ideal load-speculation (i.e. always load speculate correctly) to the realistic stride-based load-speculation mechanism. In fact, when using realistic load-speculation, it is apparent that d-collapsing provides the biggest portion of the performance gain. This leads us to study load-speculation in more detail to determine means of capturing more of the ideal performance.

5.2. The effect of pointers on load-speculation performance

Given that the load-speculation mechanism is based on detecting strides, we decided it would be useful to divide the benchmarks into two categories, namely *pointer chasing* benchmarks and *non pointer chasing* benchmarks – under the assumption that the stride-based load-speculation mechanism would be relatively ineffective for the pointer chasing benchmarks. A program was classified as pointer chasing if a large fraction of its loads are indirect. We categorized the benchmarks by profiling and inspection of code. The pointer chasing benchmark set consists of the benchmarks go and li and the other set contains the remaining benchmarks. Figures 4 - 7 report the overall IPC and speedups for the two sets.

The following can be observed regarding pointer chasing benchmarks (Figures 4 and 5):

- With ideal load-speculation (point E), the speedup is about the same as when all the benchmarks are considered. However, the performance drop with realistic load-speculation is more pronounced than in the complete set of benchmarks.
- Furthermore, for issue widths 4, 8, 16 and 32, realistic load-speculation has a very small performance effect (speedups of 5%-9%, see B in Figure 5).
- The d-collapsing gains are smaller as compared to the means for all benchmarks (points C in Figures 3 and 5). This can be attributed to the instruction mix of those benchmarks that consist of non-collapsible instructions (for 022.li, 7% of the instructions are calls and returns) and higher branch misprediction rates (go has branch prediction rate of 83.6%).

The most important observation regarding realistic load-speculation for pointer chasing benchmarks is that by itself (with no d-collapsing) it provides negligible performance gains. This is understandable because the load-speculation mechanism is based on stride prediction, and it does not perform well with pointer-based codes. Special load units for pointer-based codes have been proposed [12], however, there is no scheme known to us that combines load-speculation mechanisms for both pointer chasing and non pointer chasing codes. It is of interest, therefore, as a future research topic to investigate load-speculation mecha-

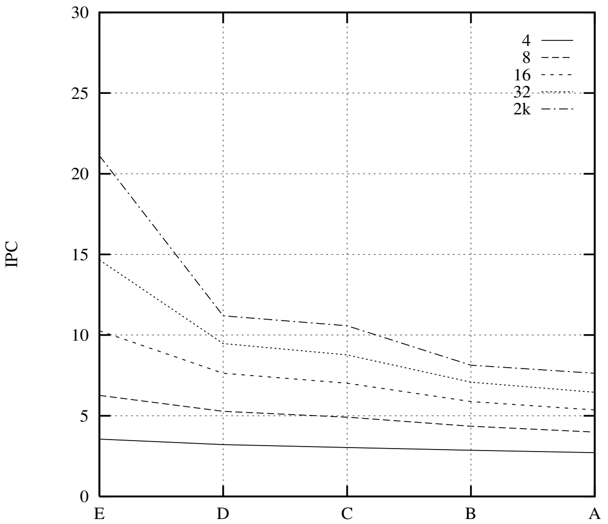

Figure 4. IPC for the "Pointer Chasing" Benchmarks

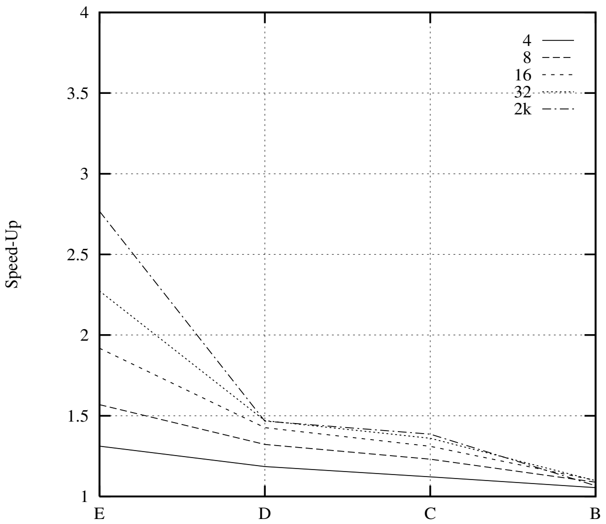

Figure 5. SpeedUp over the Superscalar Base Machine (A) for the "Pointer Chasing" Benchmarks

nisms that can provide satisfactory performance for both non-pointer and pointer chasing benchmarks.

To better understand the issues caused by pointer chasing we investigated in more detail the performance of the load-speculation mechanism we used. Table 3 suggests the following.

- For pointer chasing benchmarks, the success rate of address prediction is low (12.4-26.7%).
- Low prediction rate is mainly because a very large percentage of loads (around 38-44%) are not predicted at all – they simply do not demonstrate stride behavior.
- On the positive side, the percentage of incorrect predictions is very small implying that the counter scheme can capture the predictability of loads. Possible variations are currently being explored to determine even more accurate confidence measurements.

The increase in the number of ready (non-speculated) loads, with increasing window size (Table 3), is attributed to a corresponding increase of collapsed instructions.

Turning to the benchmarks that do not rely on pointer chasing, we have plotted the expected IPC and speedups over the base superscalar machine in Figures 6 and 7. The following can be observed:

- The overall speedups are higher when using realistic load-speculation than for the entire mix of benchmarks.
- Collapsing with realistic load-speculation (point D in Figure 7 for windows of 4, 8, 16 and 32) speedups between 1.23–1.8 are reached.

Issue Width	Ready (%)	Predicted Correctly (%)	Predicted Incorrectly (%)	Not Predicted (%)
4	30.16	26.73	4.76	38.34
8	35.94	20.87	5.16	38.02
16	39.79	15.13	5.13	39.95
32	40.29	12.42	5.35	41.94
2k	36.20	14.90	5.41	43.50

Table 3. Load-Speculation Behavior for Pointer Chasing Benchmarks with Configuration D

- The difference between speedups with ideal and realistic load-speculation is smaller when compared with the pointer chasing programs.
- The major contribution is still due to d-collapsing, but a significant contribution is made by load-speculation.

Table 4 is the counterpart of Table 3 for the non pointer chasing benchmarks. The behavior of the loads in Table 4 in comparison to the behavior of the loads for pointer chasing benchmarks (Table 3) indicates a significant reduction in the number of mispredicted loads and similar reduction of the number of not predicted loads. A significant number of loads remain not predicted, nevertheless. These results suggest that even the stride-based load-speculation method can be improved. We intend to investigate such approaches as part of our continuing research.

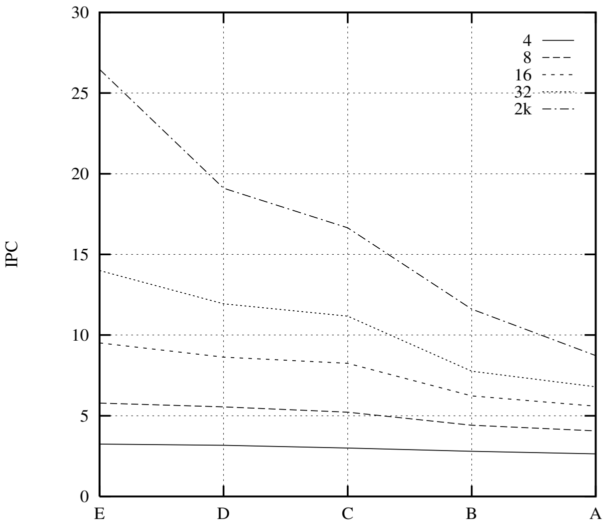

Figure 6. IPC for the non "Pointer Chasing" Benchmarks

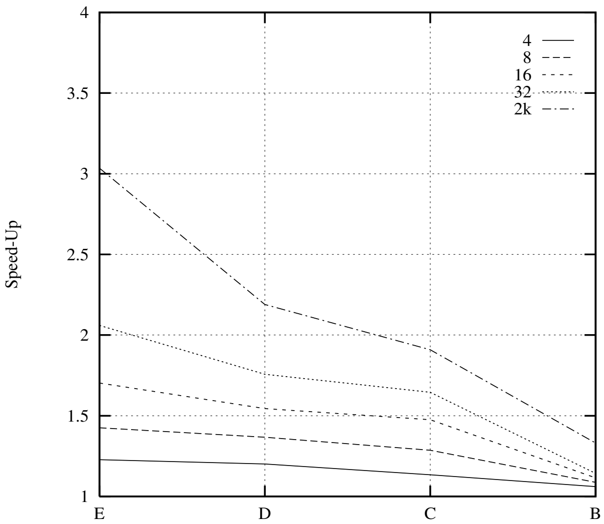

Figure 7. SpeedUp over the Superscalar Base Machine (A) for non "Pointer Chasing" Benchmarks

Issue Width	Ready (%)	Predicted Correctly (%)	Predicted Incorrectly (%)	Not Predicted (%)
4	20.67	56.95	2.17	20.21
8	39.23	39.99	1.91	18.87
16	49.94	28.16	1.97	19.93
32	47.80	30.13	1.96	20.11
2k	38.13	39.85	2.02	20.00

Table 4. Load-Speculation Behavior for non-Chasing Pointer for Configuration D

5.3. A Closer Look at D-Collapsing

As indicated earlier, d-collapsing plays an important role in the speedup gains over a base superscalar machine. To analyze in more depth the various contributions and behavior of such mechanisms we plot in Figures 8 - 10: the percentage of instructions that have been collapsed in all benchmarks, the contribution of the 3-1, 4-1, and 0 operand detection, and distance between collapsed instructions. The following conclusions are suggested by the figures.

- A large percentage of the instructions are collapsed (from 29-47% with increasing issue width - Figure 8).
- The biggest collapsing contributor is the 3-1 (Figure 9). For example, with issue windows equal to or less than 32 instructions, 3-1 collapsing accounts for 65-82% of the d-collapsing instructions.
- 4-1 collapsing mechanisms account for 13-30% of the collapsed instructions. This indicates that 4-1 collapsing may hold important performance benefits and the design of such units may need to be considered. However, doing this without adding latency will be a major issue. The results also suggests that collapsing greater than 4-1 dependences may offer very little performance benefit.
- The zero value operand detection has a contribution of 5–10%. If the implementation of such a mechanism is not expensive it could offer some performance improvements.
- An interesting statistic is the distance among the instructions that are collapsed (Figure 10). For issue widths larger than 8 instructions the majority of the collapsed instructions are not consecutive. This suggests that a wide-issue implementation will need to consider non-consecutive instruction collapsing in order to exploit the available performance gains. We also observe most of the instructions collapsed have a distance of less than 8 instructions even for the 2k issue width. This indicates that we may not need to implement across basic blocks since the average basic block size is expected to be around 6 - 8 instructions. The performance pay off versus the complexity of collapsing across basic blocks is currently being investigated.

In Tables 5 and 6 we show the average for the most frequently collapsed 3-1 and 4-1 dependence sequences for the D configuration. The following encodings are used in Tables 5 and 6, the instruction types are ar: arithmetic, lg: logic, sh: shift, mv: move, ld: load, st: store, brc: conditional branch; the source operand types are r: register, i: imme-

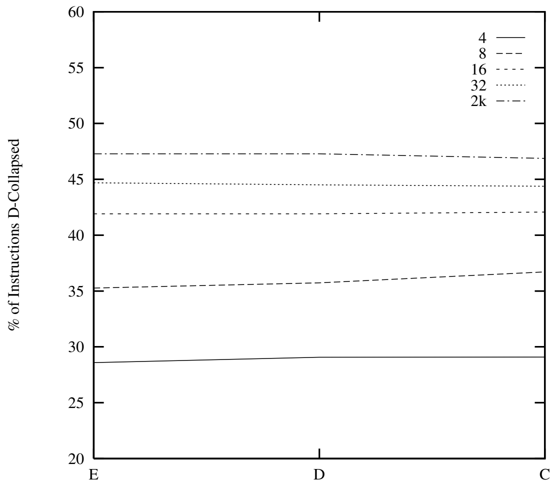

Figure 8. Instructions D-Collapsed

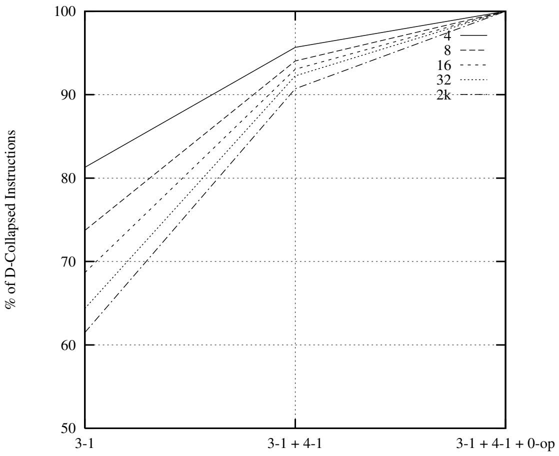

Figure 9. Contribution of the three Collapsing Mechanisms for the D Configuration

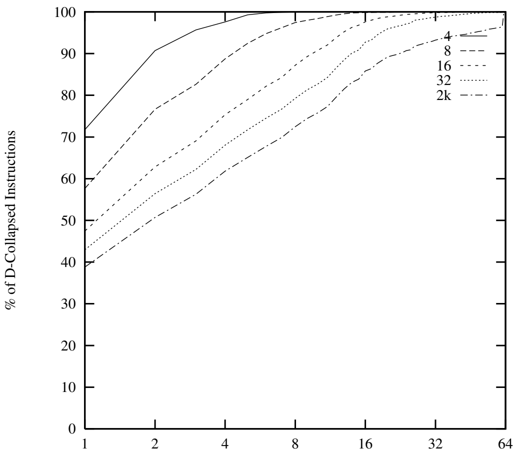

Figure 10. Distance between D-Collapsed Instructions for the D Configuration

diate, 0: zero immediate or zero register. For example, the collapsing of Rb = Rd + Rh and Rg = Rb + 8 to Rg = Rd + Rh + 8, corresponds to an arrr – arri collapsed pair. The average for each 3-1(4-1) dependence sequence is the sum of all such collapsed dependences over the total number of 3-1(4-1) collapsed dependences in all benchmarks.

6. Conclusions

In this paper we considered two hardware based schemes that enable the elimination of true data dependences: **d-speculation** and **d-collapsing**. The separate and combined use of the mechanisms that implement the different schemes was investigated. Trace driven simulation was the means used for demonstrating the potential of these methods. Several experiments were conducted for a number of machine configurations and issue widths.

The results show consistently that a substantial increase in performance is possible by using d-collapsing and realistic load-speculation (configuration D). We observed that d-collapsing is responsible for the majority of the gains though d-speculation still provides significant gains for non-pointer chasing programs. It is also established that there are large benefits to be gained if the load-speculation scheme is improved.

With regard to the behavior of d-collapsing, a large fraction of instructions, 29% to 47%, can potentially be collapsed. The number of collapsed instructions increases as windows grow larger. The distance between collapsed instructions increase marginally even for larger windows, but is usually less than 8.

Overall, the results are quite promising and suggest that data dependence speculation and collapsing can potentially have a significant impact on performance. This study reveals interesting information about the behavior of dependences that can be of value during the implementation of data speculating and collapsing mechanisms. Further research is currently undertaken in several directions that include: development of mechanisms to support data dependence speculation and collapsing, determination of ways to use compilers to increase ILP under this paradigm, and additional performance studies for more realistic environments.

7. Acknowledgements

This work was supported in part by NSF Grants MIP-9505853 and MIP-9307830 and by the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-

Operation Types		Issue Width				
Op1	Op2	2k	32	16	8	4
arrr	brc	12.68	12.67	12.25	13.62	17.08
arri	brc	12.37	12.72	12.87	14.00	15.71
arri	arri	8.04	7.12	7.01	3.35	1.75
arr0	brc	7.13	7.47	7.76	8.95	10.05
shri	ldrr	5.07	5.42	5.43	5.31	5.75
mvi	lgri	4.99	4.81	4.62	4.34	3.96
mvi	ldri	4.87	4.88	4.80	4.77	4.07
arrr	arrr	4.43	3.80	3.78	4.10	3.31
arri	ldrr	3.16	3.54	4.66	3.31	1.56
shri	arrr	3.01	3.90	3.94	4.54	4.07
arrr	shri	2.77	2.38	2.24	2.46	3.32
arri	arrr	2.02	2.37	3.56	4.81	4.38

Table 5. Collapsed 3-1 Dependences

Operation Types			Issue Width				
Op1	Op2	Op3	2k	32	16	8	4
arri	arri	arri	17.97	14.55	7.27	0.46	0.00
lgr0	lgr0	arrr	6.59	6.52	8.32	10.89	0.50
arri	arri	ldrr	6.19	7.25	4.38	0.02	0.00
arrr	arrr	arrr	5.97	6.05	5.56	4.09	2.38
arrr	shri	arrr	4.84	4.73	5.86	7.83	13.70
arri	arri	arrr	4.16	4.57	3.88	0.08	0.02
lgri	shri	ldrr	3.37	4.12	5.25	7.80	7.72
arrr	arrr	shri	3.15	2.76	3.68	4.33	5.63
shri	arrr	shri	2.88	4.39	5.66	8.28	12.17
lgrr	lgrr	lgrr	2.51	2.87	3.28	3.13	3.20
shri	arrr	ldrr	2.22	2.50	0.82	0.76	0.15
shri	arrr	ldri	2.16	2.49	3.17	4.47	5.57
shri	arrr	arrr	1.72	3.04	5.41	8.00	12.55

Table 6. Collapsed 4-1 Dependences

95-C-0127 and ARPA order no. D346. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U. S. Army Intelligence Center and Fort Huachuca, or the U.S. Government.

The authors would like to thank Rajiv Jain and Andreas Moshovos for their helpful suggestions and constructive critique while this work was in progress.

References

[1] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 342–351, May 1992.

[2] T. M. Austin and G. S. Sohi. Zero-cycle loads: Microarchitecture support for reducing load latency. In *Proceedings of the 28th Annual ACM/IEEE International Symposium and Workshop on Microarchitecture*, pages 82–92, June 1995.

[3] M. Butler, T.-Y. Yeh, Y. N. Patt, M. Alsup, H. Scales, and M. Shebanow. Single instruction stream parallelism is greater than two. In *Proceedings of the 18th International Symposium on Computer Architecture*, May 1991.

[4] T. F. Chen and J. L. Baer. Effective hardware-based data prefetching for high performance processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[5] R. J. Eickemeyer and S. Vassiliadis. A load instruction unit for pipelined processors. *IBM Journal of Research and Development*, 37(4):547–564, July 1993.

[6] N. P. Jouppi and D. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

[7] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 46–57, May 1992.

[8] J. R. Larus. Efficient program tracing. *IEEE Computer*, 26(5):52–61, May 1993.

[9] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and data speculation. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[10] N. Malik, R. J. Eickemeyer, and S. Vassiliadis. Interlock collapsing alu for increased instruction-level parallelism. In *Proceedings of the 25th Annual ACM/IEEE International Symposium and Workshop on Microarchitecture*, September 1992.

[11] S. McFarling. Combining branch predictors. In *DEC WRL TN-36*, June 1993.

[12] S. Mehrotra and L. Harrison. Examination of a memory access classification scheme for pointer intensive and numeric programs. In *Proceedings of the 10th International Conference on Supercomputing*, May 1996.

[13] S. MICROSYSTEMS. *The SPARC Architecture Manual*. Prentice Hall, 1992.

[14] R. K. Montoye, E. Hokenek, and S. L. Runyon. Design of the ibm risc system/6000 floating-point execution unit. *IBM Journal of Research and Development*, 34(1):59–70, January 1990.

[15] A. Moshovos. Increasing instruction level parallelism through instruction coallescing. *private communication*, 1995.

[16] R. R. Oehler and R. D. Groves. Ibm risc system/6000 processor architecture. *IBM Journal of Research and Development*, 34(1):23–36, January 1990.

[17] J. Phillips and S. Vassiliadis. High performance 3-1 interlock collapsing alu's. *IEEE Transactions on Computers*, 43(3):257–268, March 1994.

[18] S. Vassiliadis, B. Blaner, and R. J. Eickemeyer. Scism: A scalable compound instruction set machine architecture. *IBM Journal of Research and Development*, 38(1):59–78, January 1994.

[19] S. Vassiliadis, J. Phillips, and B. Blanner. Interlock collapsing alu's. *IEEE Transactions on Computers*, 42(7):825–839, July 1993.

[20] D. W. Wall. Limits of instruction level parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, April 1991.

[21] S. Weiss and J. E. Smith. *Inside IBM Power and PowerPC*. Morgan Kaufmann Publishers Inc., San Mateo, CA, 1994.