Exploiting Basic Block Value Locality with Block Reuse

Jian Huang
Department of Computer Science and Engineering
Minnesota Supercomputing Institute
University of Minnesota, Minneapolis, MN 55455
huangj@cs.umn.edu

David J. Lilja
Department of Electrical and Computer Engineering
Minnesota Supercomputing Institute
University of Minnesota, Minneapolis, MN 55455
lilja@ece.umn.edu

Abstract

Speculative execution and instruction reuse are two important strategies that have been investigated for improving processor performance. Value prediction at the instruction level has been introduced to allow even more aggressive speculation and reuse than previous techniques. This study suggests that using compiler support to extend value prediction and reuse to a coarser granularity, such as a basic block, may have substantial performance benefits. We investigate the input and output values of basic blocks and find that these values can be quite regular and predictable. For the SPEC benchmark programs evaluated, 90% of the basic blocks have fewer than 4 register inputs, 5 live register outputs, 4 memory inputs and 2 memory outputs. About 16% to 41% of all the basic blocks are simply repeating earlier calculations when the programs are compiled with the -O2 optimization level in the GCC compiler. Compiler optimizations, such as loop-unrolling and function inlining, affect the sizes of basic blocks, but have no significant impact on their value locality. Based on these results, we evaluate the potential benefit of basic block reuse using a novel mechanism called a block history buffer. This mechanism records input and live output values of basic blocks to provide value prediction and reuse at the basic block level. Simulation results show that using a reasonably-sized block history buffer to provide basic block reuse in a 4-way issue superscalar processor can improve execution time for the tested SPEC programs by 1% to 14% with an overall average of 9% with reasonable hardware assumptions.

Keywords: *block history buffer, block reuse, compiler flow analysis, value locality, value reuse*

1 Introduction

Dependences between instructions limit the instruction execution rate of a typical superscalar processor to an average of only about 1.7 to 2.1 instructions per cycle (IPC) [16]. Speculative execution and multithreading are two techniques that have been introduced to extend the limits of instruction-level parallelism. Some recent examples include the multiscalar architecture [10], the trace processor (TP) [2], the superthreading architecture [3], the multiprocessor-on-a-chip (MOAC) [4], and the superspeculative processor (SSP) [9]. The multiscalar and trace processor architectures advocate a wide-issue multi-threaded approach, while the MOAC incorporates multiple separate processors on a single chip. The supertheaded processor is a hybrid of superscalar and multithreaded architectures that speculates on control dependences while resolving data dependences at runtime. The TP and SSP, on the other hand, speculate on both control and data dependences, while the MOAC incorporates only data speculation [12].

To speculate beyond control and data dependences, Lipasti *et al* [5] introduced the concept of *value locality* for each instruction. This locality is a measure of how often an instruction regenerates a value that it has produced before. They discovered that the values produced by an instruction are actually very regular and predictable. Tyson and Austin [14] further found that 29% of the load instructions in the SPECint benchmarks and 44% of the loads in the SPECfp benchmarks reload the same value as the last time the load was executed. *Value locality* allows processors to predict the actual data values that will be produced by instructions before they are executed.

Several techniques have been proposed to improve value prediction accuracy. These include a history-based predictor, a stride-based predictor, a hybrid predictor [6], and a context-based predictor [7]. All of these schemes work at the level of a single instruction, and try to predict the next value that will be produced by an instruction based on the previous values already generated. Since these schemes try to cache as large a history of values as possible, they require large hardware tables on the processor die.

The scope of all these techniques can be too limited, however, and the values predicted can be wrong. By determining actual values instead of simply predicting them, the processor could throw away redundant work and simply jump directly to the next task. For example, the dynamic instruction reuse proposed by Sodani and Sohi [11] saves the input and output register values for each instruction to allow the execution of the instruction to be skipped when the current input values match a previously cached set of values. We observe, however, that the inputs and outputs of a chain of instructions are highly correlated. Thus, a natural coarsening of the granularity for value reuse is the basic block. A basic block can be viewed as a *superinstruction* that has some set of inputs and produces some set of live output values. Using the basic block as the prediction and reuse unit may save hardware compared to previous instruction-level reuse and prediction schemes in addition to reducing execution time.

In this work, we investigate the input and output value locality of basic blocks to determine their predictability and their potential for reuse. In the following experiments, the basic block boundaries are determined dynamically at run-time. The upward-exposed inputs of each basic block, as well as its live outputs, are stored in a new hardware mechanism called a *block history buffer*. The processor uses these stored values to determine the output values a basic block will produce the next time it is executed. If the current inputs to a block are found to be the same as the last time the block was executed, all of the instructions in the block can be skipped. This technique is called *block reuse* in contrast to instruction reuse [11]. In order to prevent the register outputs that are dead after a block's execution from occupying limited *block history buffer* resources, and to prevent dead outputs from poisoning a block's value locality, we use the *GCC* compiler to mark dead register outputs, and pass this information to the hardware. Our simulation results show that block reuse can boost performance by 1% to 14% over existing superscalar processors with reasonable hardware assumptions.

In the remainder of the paper, Section 2 defines and quantifies the concepts of input and output value locality for basic blocks. Section 3 describes the idea of block reuse, the hardware implementation of the *block history buffer*, and evaluates the performance potential of block reuse. Related work is described in Section 4 and Section 5 concludes the work.

2 Input and Output Value Locality of Basic Blocks

Each instruction in a program belongs to a basic block, which is a sequence of instructions with a single entry and a single exit point [1]. Instructions within a basic block are correlated in that some inputs to an instruction may be produced by previous instructions within the same block. An input which is not produced within the same block is called an *upward-exposed input*. The set of all upward-exposed inputs compose the input set of a basic block. This set includes both registers and memory references. When a basic block is executed a second time and the set of input values are the same as the last time the block was executed, we say that this block is demonstrating *block-input value locality*. *Block-output value locality* is defined similarly. However, some values produced inside a basic block may not be needed by the following blocks, since they may be either unused or overwritten by the following blocks in the execution path. These types of outputs are termed *dead outputs*, similar to the concept of a dead definition in

Metric	Alvinn	Compress	Ear	Go	Ijpeg	Li	M88Ksim	Perl	Wordcount
Arithmetic Mean(AM)	4.82	4.35	4.81	5.72	5.95	4.15	4.68	4.15	4.14
Weighted Mean(WM)	8.89	9.80	10.09	6.03	14.03	4.00	4.16	4.76	3.35
Number of Blocks (NB)	1071	760	1632	8969	2755	1462	2388	3285	487

Table 1: Average size and number of basic blocks for the test programs. WM uses block execution frequency as the weight, while AM is the average of the static block counts.

a compiler [1]. All outputs that are used outside a basic block are called its *live outputs*. The output value locality of a block refers only to its live outputs. Instructions also have input and output value locality [5]. The input and output value locality of a block that has only a single instruction is the same as that instruction's value locality. We use the terms input and output value locality in later discussions to refer to block input and output value locality.

In this study, we construct basic blocks and their input and output sets dynamically at runtime as discussed in Section 3. We store up to four sets of input and output values for a block from its previous four executions. The values that were read or produced by the immediately previous execution of the block are called its *depth-1 inputs* or *outputs*. The values that were read or produced by this block in *either* of the two previous executions are called the *depth-2 inputs* or *outputs*. *Depth-n inputs* or *outputs* are defined accordingly. The value locality corresponding to depth-n inputs or outputs is called *depth-n input* or *output value locality*. All programs are compiled with the *GCC* compiler using the *-O2* flag.

2.1 Characteristics of Basic Block Inputs and Outputs

A basic block can consist of an arbitrary number of instructions, although typical values range between 1 and 25. Table 1 shows the average number of instructions in a basic block for a collection of the SPEC benchmark programs. The corresponding cumulative execution frequencies are shown in Figure 1. We see that for 5 of the 9 programs, approximately 70 percent of the blocks have no more than 5 instructions. For 6 out of the 9 programs, 90 percent of the blocks have fewer than 15 instructions. For most programs, roughly 10 percent of the basic blocks have only 1 instruction, and fewer than 5 percent have more than 20 instructions. For *Ijpeg* and *Ear*, however, about 15 to 20% of the blocks have more than 20 instructions.

Since most of the basic blocks are not very large, we expect to see relatively few inputs and outputs for each

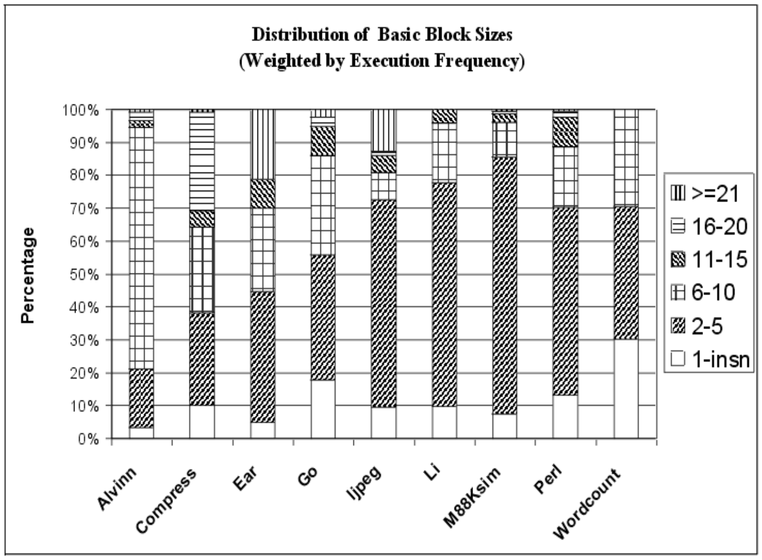

Figure 1: Distribution of executed instructions for different basic block sizes.

block. As shown in Figure 2, roughly 90 percent of the blocks have fewer than 4 upward-exposed register inputs and fewer than 4 memory inputs for all programs except *Ear*. We have modified the *GCC* compiler to mark the dead register outputs in each instruction using the SimpleScalar [8] instruction annotation tool. The hardware interprets this information to exclude the marked registers from the set of live outputs of each basic block. From this analysis, we find that the number of live register outputs in a block tends to be slightly larger than the number of inputs, as shown in Figure 2. About 90 percent of the basic blocks have fewer than 5 live register outputs. Roughly 10 to 15 percent of the basic blocks have no live register outputs, which is very close to the percentage of blocks that contain only 1 instruction. Usually these single-instructions basic blocks contain only a single branch or jump instruction.

The number of memory outputs per basic block is very small, due to the infrequent appearance of store instructions. Most of the values written to memory are used by later basic blocks. Hence, we assume all of the memory writes are live. We find that 85 to 95 percent of the basic blocks have at most 1 store, while 25 to 75 percent of all blocks actually have no stores at all.

The static arithmetic mean for the number of block inputs and outputs, as well as the mean weighted by a block's execution frequency, are shown in Table 2. All programs except *Wordcount* have a larger weighted mean

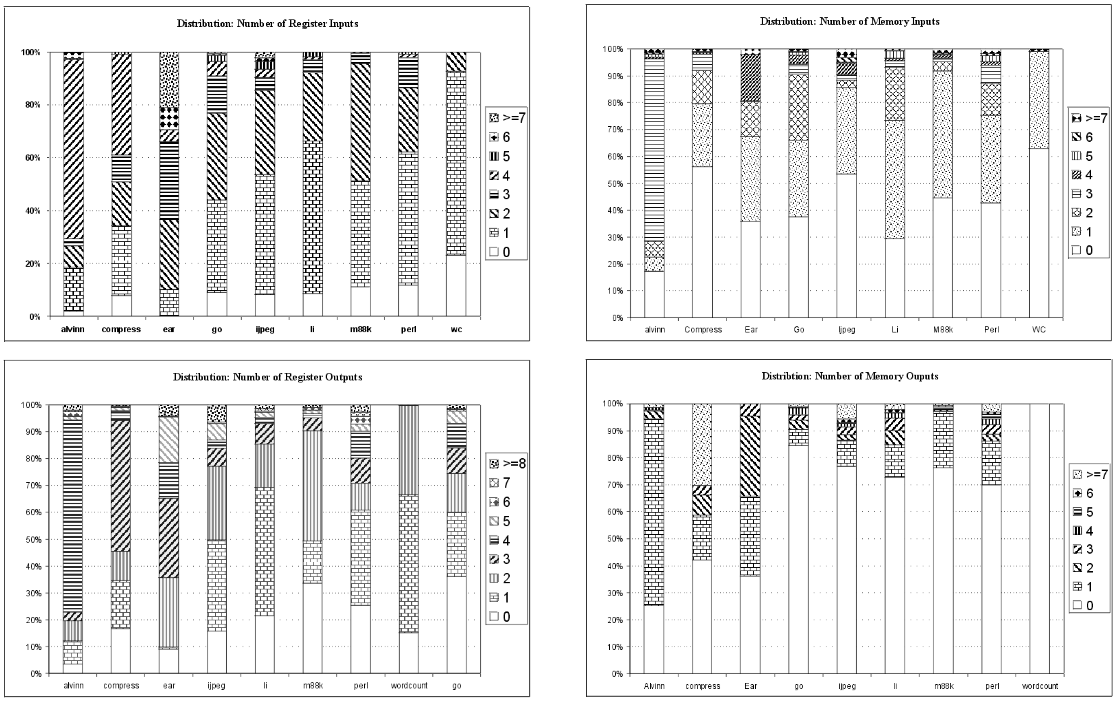

Figure 2: Distribution of the number of inputs and outputs for a basic block weighted by execution frequency.

than arithmetic mean. This difference is especially large for *Alvinn, Compress*, *Ear* and *Go*. It indicates that the frequently executed blocks in these programs have a larger number of inputs and outputs than the "typical" block.

2.2 Ideal Value Locality of Basic Blocks

The relatively small numbers of inputs and outputs in a basic block provide the basis for the *block history buffer* mechanism to store this information (described in Section 3.1). If the input values of basic blocks tend to recur,

Metric	Alvinn	Compress	Ear	Go	Ijpeg	Li	M88Ksim	Perl	Word-count
Mean Reg-In	1.40	1.30	1.40	1.40	1.40	1.20	1.20	1.10	1.20
WM Reg-In	3.38	2.47	3.96	1.85	1.74	1.38	1.42	1.42	0.84
Mean Reg-Out	2.50	2.10	2.40	1.70	2.10	2.00	1.90	1.60	2.10
WM Reg-Out	3.67	2.13	3.23	1.62	2.10	1.44	1.39	1.94	1.18
Mean Mem-In	1.00	0.90	1.00	1.10	1.30	0.90	0.90	1.00	0.90
WM Mem-In	2.54	0.72	1.59	1.21	1.87	1.18	0.72	1.22	0.36
Mean Mem-Out	0.60	0.60	0.60	0.50	0.90	0.70	0.60	0.60	0.60
WM Mem-Out	0.92	5.24	1.03	0.45	1.12	0.70	0.36	0.85	0.00

Table 2: Arithmetic and execution frequency weighted mean of the number of inputs and outputs for basic blocks.

a considerable amount of redundant work can be avoided by simply retrieving the stored output values when the input values match, and thereby avoiding the need to re-execute all of the instructions in the block.

How repetitive or determinable are a block's input values? To answer this question, we studied the behavior of 9 SPEC benchmark programs using the SimpleScalar tool set [8]. We first assume unlimited hardware resources and record all of the input and output values for all basic blocks. The basic blocks themselves are constructed on the fly at run-time with a value history of depth one to four stored for each block. This information is adequate to summarize the overall determinability. The *depth-n input value locality* for each block is calculated as the number of times a block finds the same input values in the depth-n input history table divided by the number of times the block is executed. Then the overall depth-n input value locality is weighted by the block's execution frequency. The overall *depth-n output value locality* is obtained in a similar fashion. The overall block value locality for the 9 SPEC programs tested are listed in Tables 3 and 4.

We find that the input value locality varies from 2.21% to 41.44%, and the output value locality ranges between 3.09% and 51.63%. *Ear* has the worst basic block value locality for both input and outputs. In this program, most of the blocks with high execution frequencies are large with many register inputs and possibly some memory inputs. Furthermore, its loops often update induction variables within frequently executed basic blocks. As a result, these blocks tend to have low input locality. From the relatively small differences between the input and output value locality numbers, we may infer that most basic blocks produce repeated outputs only when they have repeated inputs. However, these differences may indicate an opportunity to predict the output values of a basic block for speculative execution.

Increasing the history depth tends not to produce a significant increase in the input or output value locality, except for *M88Ksim*, which is a processor simulator with a well-defined input domain (a fixed instruction set). For the other programs, the set of inputs to a basic block has a large domain, so that even a tiny change in any input values will cause the basic block to lose its value locality. As a result, individual basic blocks tend to exhibit either very good value locality or almost none at all. A depth-one history is sufficient to capture most of the essential value locality behavior of a block for most of the tested programs. Consequently, if the goal is simply to identify redundant basic block executions, a history depth of one is adequate.

Depth/programs	Alvinn	Compress	Ear	Go	Ijpeg	Li	M88Ksim	Perl	Wordcount
1	16.21	19.68	2.21	28.80	21.16	30.52	39.68	41.44	32.84
2	20.22	20.54	2.42	34.43	22.33	32.96	53.54	44.52	33.57
3	21.80	21.20	2.45	38.94	23.54	34.83	54.89	48.24	34.15
4	23.85	21.65	2.81	42.33	24.36	35.62	69.24	49.42	34.52

Table 3: Input value locality for different history depths (percentage).

Depth/programs	Alvinn	Compress	Ear	Go	Ijpeg	Li	M88Ksim	Perl	Word-count
1	18.85	23.47	3.15	46.27	31.90	39.70	51.63	42.26	37.40
2	21.59	24.10	3.18	50.65	33.62	44.58	64.95	45.92	37.41
3	23.19	24.79	3.29	54.50	35.03	50.66	65.77	51.06	37.41
4	24.71	25.20	3.78	57.62	36.14	59.96	80.58	52.15	37.41

Table 4: Output value locality for different history depths (percentage).

2.3 Determinable Locality

The unlimited resource assumption of the previous subsection will not help us to understand the potential benefits of exploiting basic block input and output value locality. We thus assume the number of input register values the hardware can store is 4, and the number of memory input values is at most 4. The corresponding numbers for outputs are 5 register values and 2 memory values. Based on the results in Figure 2, these parameters are sufficient to cover the requirements of 90% of all of the basic blocks. If the hardware configuration is too small to store all inputs and outputs of a block, it must be assumed that there is no value locality. The updated locality values with this resource limitation are shown in Table 5. The block value locality observed in this case is quite close to the unlimited resource case, suggesting that we may be able to exploit basic block value locality with realistic hardware configurations.

An actual processor could not store the necessary input and output values for all basic blocks all the time. This

	Alvinn	Compress	Ear	Go	Ijpeg	Li	M88Ksim	Perl	Word-count
Input Locality	13.85	19.08	1.87	27.37	17.91	29.00	28.67	34.33	32.83
Output Locality	17.01	23.16	2.87	45.60	34.35	39.50	57.66	41.92	37.40

Table 5: Block value locality with a limited number of input and output values stored (percentage).

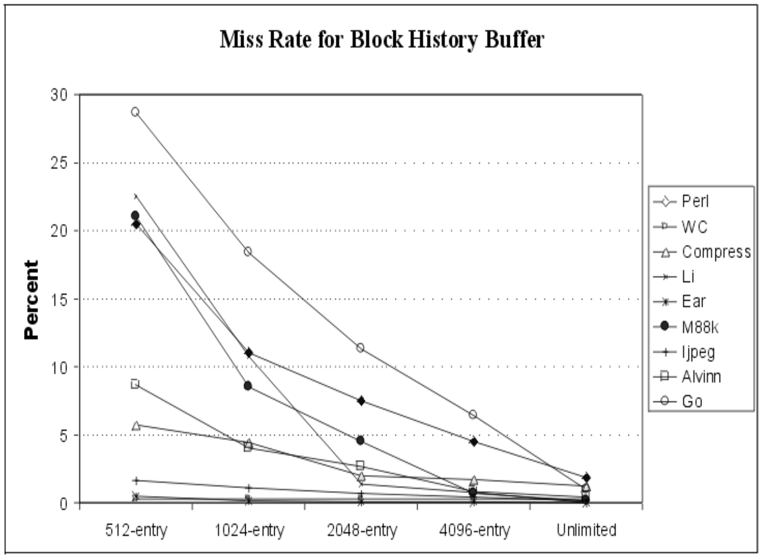

Figure 3: Miss rates for different sizes of the block history buffer.

is too expensive and practically impossible. Instead, we next examine the effect on value locality of restricting the number of entries that are stored. Since this buffer acts like a value history buffer for basic block data, we call it a *block history buffer*. This *block history buffer* is indexed with the address of the first instruction in a basic block, shifted right 2 bits. We evaluated history buffer sizes of 512, 1024, 2048 and 4096 entries. The miss rate for each configuration is plotted in Figure 3. *Word-count, Compress* and *Alvinn* have the smallest number of basic blocks (see Table 1) and consequently have the lowest miss rates in the *block history buffer*. For large programs, such as *Perl* and *Go*, the miss rates are 20.50% and 28.63% when the buffer is small, and 4.51% and 6.37% when the buffer is as large as 4K entries. The input and output value locality for these programs is still substantial under the above resource limitations, as shown in Tables 6 and 7. Observe that a buffer size of 2K entries is sufficient to cover the block execution window for most programs. Even *Go*, which has 8969 unique basic blocks, has a miss rate of only 11.28% with the 2K buffer. Hence, we decide to use the 2048-entry configuration in all of the subsequent experiments.

Programs	512-entry	1024-entry	2048-entry	4096entry	Unlimited
Alvinn	8.40	10.93	11.87	13.28	16.21
Compress	17.84	18.35	18.72	18.84	19.68
Ear	1.65	1.73	1.75	1.76	2.21
Go	18.43	21.23	23.47	24.87	28.80
Ijpeg	13.96	14.16	14.28	14.42	19.64
Li	18.88	24.16	30.13	28.44	30.50
M88k	18.45	24.10	25.73	27.85	31.09
Perl	25.93	30.03	31.43	32.27	41.44
WC	32.82	32.83	32.83	32.83	32.84

Table 6: Input value locality (percentage) with block history buffer storage limits.

Program	512-entry	1024-entry	2048-entry	4096-entry	Unlimited
Alvinn	17.85	18.49	18.54	18.85	18.85
Compress	23.06	23.24	23.40	23.42	23.47
Ear	3.13	3.14	3.15	3.15	3.15
Go	42.48	43.98	44.91	45.42	46.27
Ijpeg	34.44	34.52	34.61	34.66	34.70
Li	38.29	38.81	39.61	39.64	39.70
M88k	55.22	56.30	57.06	58.04	58.05
Perl	40.29	41.12	41.52	41.83	42.26
WC	37.40	37.40	37.40	37.40	37.40

Table 7: Output value locality (percentage) with block history buffer storage limits.

2.4 The Effect of Compiler Optimizations on Basic Block Value Locality

The previous experiments all used *GCC*'s *-O2* optimization level. This optimization level does not include loop-unrolling or function inlining, which could possibly change the size of the basic blocks, as well as their input and output value localities. *Loop-unrolling* makes the loop body larger by merging a few iterations of the original loop into a single iteration, so that fewer total iterations are executed, but each iteration is larger. This change may also increase the size of the program's basic blocks. *Function- inlining* reduces the number of function calls by replacing the call with a copy of the function body. By eliminating the call and return instructions, this optimization could also increase the average basic block size.

Tables 8 to 11 show the effects of applying each optimization individually, and together. When inlining is used, there are no significant changes in the basic block sizes for most of the programs. Similarly, there is little change in the corresponding sizes of the input and output sets. However, *Ijpeg* and *Alvinn* are the exceptions. For *Ijpeg*, the weighted average size of the basic blocks reduces considerably while the number of register inputs per block increases by 30 percent. The number of memory inputs, live register outputs, and memory outputs drops by 15 percent, however. For *Alvinn*, the number of live register outputs drops by 40 percent while the other metrics remain approximately the same. Furthermore, inlining improves the execution time noticeably for *Perl, Li,* and *Ijpeg*, but it degrades significantly for *Alvinn, M88Ksim*, and *Go*. Inlining has little effect on the blocks' input and output localities.

With loop-unrolling, the average basic block size, and the number of register inputs, live register outputs, memory inputs and memory outputs, increase significantly for most programs. Since loop-unrolling merges a few iterations of the original loop into one iteration, the total number of upward-exposed inputs and live outputs could be reduced compared to the sum of the number of inputs and outputs of the iterations before merging. In fact, this effect is prominent for *Alvinn* and *Ear*, where the average number of register inputs and live register outputs dropped by a large margin. The input and output locality also improve considerably for *Alvinn, Compress, Ear*, and *Wordcount*. The value locality does not change much for the rest of the programs, though. For *Perl, Li, Ear*, and *Alvinn*, the overall execution time improves with loop-unrolling, but for the other programs, the execution time actually degrades significantly.

Options	Alvinn	Compress	Ear	Go	Ijpeg	Li	M88Ksim	Perl	Wordcount
Register Inputs									
O2 only	3.38	2.47	3.96	1.85	1.74	1.38	1.42	1.42	0.84
in-line	3.38	2.47	3.96	1.86	2.28	1.38	1.42	1.43	0.84
unroll	1.79	3.47	3.79	1.83	1.77	1.38	1.51	1.40	0.99
both	1.79	3.47	3.79	1.84	2.25	1.38	1.51	1.40	0.99
Memory Inputs									
O2 only	2.54	0.72	1.59	1.21	1.87	1.18	0.72	1.22	0.36
in-line	2.54	0.72	1.59	1.18	1.56	1.21	0.71	1.22	0.36
unroll	2.09	1.01	2.34	1.21	1.97	1.18	0.90	1.21	0.47
both	2.09	1.01	2.33	1.18	1.93	1.21	0.89	1.22	0.47

Table 8: Mean number of register and memory inputs (weighted) for different compiler optimizations

Options	Alvinn	Compress	Ear	Go	Ijpeg	Li	M88Ksim	Perl	Wordcount
Register Outputs									
O2 only	3.67	2.13	3.23	1.62	2.10	1.44	1.39	1.94	1.18
in-line	2.20	2.11	3.23	1.60	1.91	1.42	1.38	1.94	1.18
unroll	1.77	3.44	2.78	1.61	2.14	1.44	1.35	1.92	1.07
both	1.77	3.40	2.78	1.59	1.90	1.42	1.34	1.93	1.07
Memory Outputs									
O2 only	0.92	5.24	1.03	0.45	1.12	0.70	0.36	0.85	0.00
in-line	0.92	5.24	1.03	0.42	0.75	0.71	0.35	0.85	0.00
unroll	1.15	7.97	1.72	0.45	1.18	0.70	0.45	0.85	0.00
both	1.15	7.97	1.71	0.42	0.95	0.71	0.44	0.85	0.00

Table 9: Mean number of register and memory outputs (weighted) for different compiler optimizations

When both optimizations are performed, the changes brought about by unrolling appear to overshadow the changes from inlining. Since these two optimizations are based on heuristics, they do not always produce the same effect for different programs. However, a possible conclusion is that loop-unrolling tends to increase the average number of memory outputs per basic block and the average size of the basic blocks. We use only the *-O2* optimization flag in the subsequent experiments due to the unpredictable effects of loop-unrolling and function-inlining.

3 The Performance Potential of Basic Block Reuse

Good input value locality for a basic block provides opportunities to improve the performance of a processor. The instruction value prediction table in a superscalar processor could be replaced with a *block history buffer* (BHB) that can be used for both value prediction and block reuse. Specifically, when the current input values to a basic

Options	Alvinn	Compress	Ear	Go	Ijpeg	Li	M88Ksim	Perl	Wordcount
O2 only	8.89	9.80	10.09	6.03	14.03	4.00	4.16	4.76	3.35
in-line	8.89	9.80	10.09	5.99	9.71	4.05	4.15	4.75	3.35
unroll	11.24	14.60	13.38	6.04	14.67	4.00	4.49	4.76	3.82
both	11.24	14.60	13.34	5.99	11.06	4.04	4.47	4.76	3.82

Table 10: Weighted mean of basic block size for different compiler optimizations

Options	Alvinn	Compress	Ear	Go	Ijpeg	Li	M88Ksim	Perl	Wordcount
Input Locality									
O2 only	16.20	19.68	2.22	28.78	19.62	30.45	31.09	41.40	32.84
in-line	16.20	20.14	2.21	28.83	17.44	31.24	31.17	41.14	32.84
unroll	44.12	31.77	6.37	29.62	21.14	30.47	39.66	42.11	40.33
Both	44.11	32.50	6.63	29.90	21.16	31.28	39.64	41.24	40.33
Output Locality									
O2 only	18.85	23.47	3.15	46.27	34.70	39.70	58.05	42.26	37.40
in-line	18.91	23.76	3.15	47.25	32.85	40.80	57.99	43.04	37.41
unroll	49.96	38.80	6.63	48.13	31.90	39.70	51.63	43.31	45.81
both	49.96	39.28	6.87	49.11	30.44	40.81	51.50	43.41	45.81

Table 11: Block input and output locality (percent) for different compiler optimizations

block are identical to those stored in the *BHB*, the stored output values can be passed to the inputs of the next basic block to be executed, thereby allowing the processor to skip the execution of all of the instructions in the current block. [1] Furthermore, when one block sees a repetition of its input values, its successors are likely to have duplicated input values in the same execution path. We call this program behavior a *flow of input value locality*. The number of basic blocks involved in a flow before a block in the sequence sees differing inputs is called the *run-length* of input value locality. When a series of blocks demonstrate input locality together, the processor can skip all of the work that is included in this series of blocks and directly update the output registers and memory. Hence, the sizes of the blocks involved in a flow are very important. We call the total number of instructions included in this type of flow of basic blocks the *Task Redundancy* (TR) of the sequence of blocks. The larger the TR, the greater the performance potential of block reuse.

The average run-length with uninterrupted input locality ranges between 1.15 and 3.65 basic blocks, but the average TR varies from 1.70 to 18.33 instructions, as shown in Table 12. The average size of the basic blocks involved in a run is larger than the average size of all basic blocks shown in Table 1. *Wordcount*, however, is a

[1] More aggressive implementations could use the history buffer to predict block output values even when the input values have changed. This speculative use of the *BHB* is beyond the scope of this paper, however.

Metrics	Alvinn	Compress	Ear	Go	Ijpeg	Li	M88Ksim	Perl	Word-count
run-length (blocks)	3.65	1.65	2.08	1.57	1.48	1.74	2.57	2.02	1.15
task redundancy	18.33	5.05	11.16	5.83	9.24	4.69	8.37	8.23	1.70

Table 12: Average run-length of input locality flow and average task redundancy (in instructions) for basic blocks.

short program that repetitively executes several switch statements, which makes it consist of many small basic blocks, as shown in Figure 1. As a result, the average size of basic blocks in the run is actually smaller than the overall average block size for *Wordcount*. The other programs typically have TR values of around 4-9 instructions. The average TR for a locality flow is large for floating point programs like *Alvinn* and *Ear*, although *Ear* exhibits little input value locality.

If the task redundancy in a program is not large enough, skipping the execution of the basic blocks cannot offset the time required to access the *BHB* and update the processor state. Figure 4 depicts the distribution of skippable instructions for different basic block sizes. About 2 to 35 percent of the executed instructions are redundant, and hence are skippable. For *Wordcount*, most of the skippable instructions belong to one-instruction basic blocks. Thus, the benefit of block reuse cannot be large for this program. *Ear* has very low input locality, and the total number of instructions that are skippable is less than 3 percent, which means block reuse will not be effective for *Ear*, either. For the other programs, skippable instructions that belong to basic blocks of 3 or more instructions comprise 5 to 28 percent of the total instructions executed. Skipping the execution of these blocks may compensate for the time required to interrogate the *BHB* and the data cache, and the time required to update the processor state, to thereby provide a performance benefit.

3.1 Hardware Implementation

To evaluate the potential performance benefit of block reuse, we propose one possible design. The input and live output values must be stored for each basic block in the *block history buffer* (BHB) along with the starting address of the next basic block. When the entry point to a block is encountered in the execution of a program, the *BHB* is checked to see if the output of this block is determinable. That is, if all of the input values to the block (including any memory inputs stored in the data cache) match the stored values in the *BHB*, the processor jumps to the subsequent block and skips all of the work in the current block. If it is not determinable, however, the processor issues

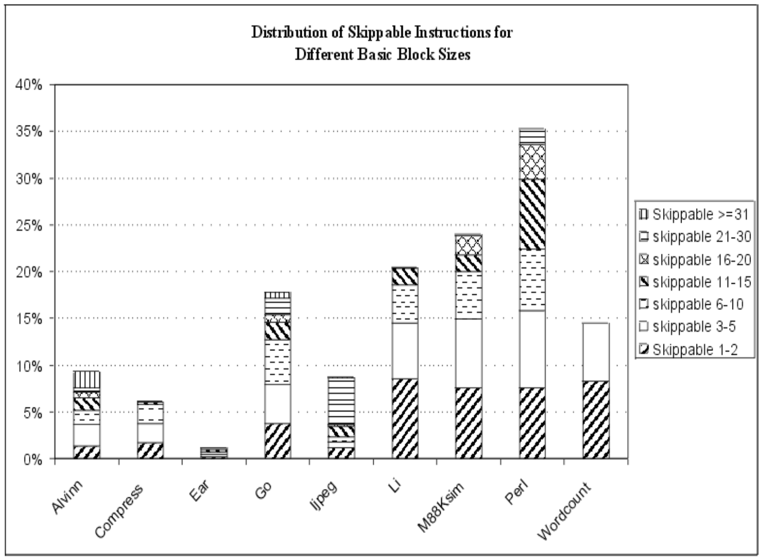

Figure 4: Distribution of skippable instructions for different block sizes.

instructions to the functional units as usual. When any instruction in a basic block commits, the *BHB* is updated. Figure 5 shows the processor model we use.

Basic blocks are constructed dynamically using the following algorithm:

1. Any instruction after a branch is identified as the entry point of a new block. The first instruction of a program is the entry point of a block automatically. Note that subroutine calls and returns are treated exactly as any other type of branch instructions.

2. Executing a branch instruction marks the end of a basic block.

3. A branch to the middle of a basic block splits the current basic block into two separate blocks. (Note, a performance optimization could duplicate the instructions after the split point to create a new block entry in the *BHB* instead of splitting the current block. We do not investigate this optimization in this paper.)

Each *BHB* entry contains the 6 fields shown in Figure 6. The *Tag* stores the starting address of a basic block. The *Reg-In* field contains several subfields. The *input mask* subfield maintains one valid bit for each logical register in the instruction set architecture and *n* sub-entries to store up to *n* actual data values. The *Reg-Out* field is organized in the same fashion. Each subentry in the *Mem-In* and *Mem-Out* fields has a *tag* that stores the program counter (PC) of the memory reference instruction, an *Addr* field that stores the memory address for the reference,

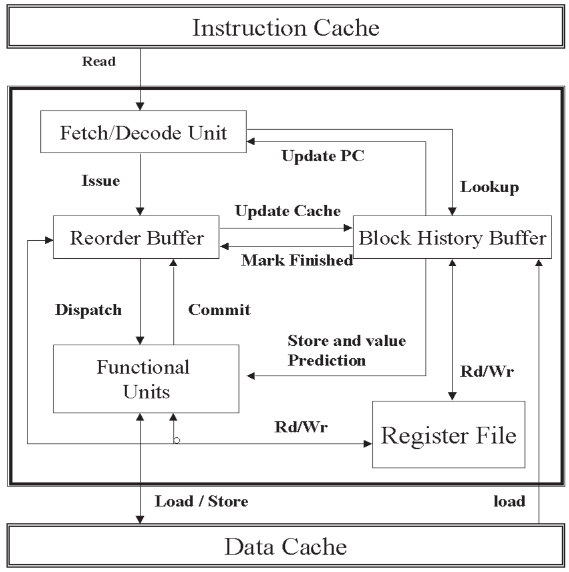

Figure 5: The processor model used for evaluating the performance potential of block reuse.

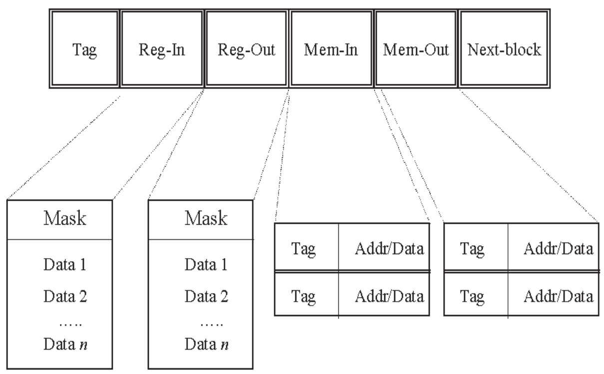

Figure 6: A possible design of an entry in the *block history buffer.*

and a *Data* field to store the actual value. Each data field has a full/empty bit to indicate if that field is currently storing a valid value. The *Next_Block* field records the starting address of the block that follows when the current block is involved in a flow of input value locality. When an instruction is fetched, the *BHB* is queried. If this instruction matches an entry for a block in the BHB, the current input values to this basic block are compared with the buffered values. When any entry in the *Mem-In* field of a basic block is valid, the data cache must be accessed. If the access produces a hit, the value from the data cache is compared with the buffered values. If the cache access is a miss, the memory contents are assumed to be different and value locality is lost.

The hardware collects the input and output values of the basic blocks dynamically. When an instruction is executed, the *input mask* bits for all logical input registers are set, and the appropriate *output mask* bits are set for the block's live output registers. Note that the registers that are live at the end of the basic block have been previously marked by the compiler. The memory input and output fields are used in a first-come-first-served manner, and the full/empty-bit is set when any entry is taken. If the *output mask* bit is set for a register that the current instruction is trying to read, this read is not an upward-exposed input. In this case, the *input mask* is left unchanged. Also, if a load instruction finds that the address it is trying to read already resides in the *Mem-Out* field, the load is not upward-exposed. Consequently the memory input field is left untouched.

When the *BHB* determines that all of the instructions in the block are redundant and can be skipped, it will perform one of the two following actions depending on the type of exception processing desired.

- For precise exceptions, the instructions are issued as in normal processing. They are marked as completed when they reserve a reorder buffer entry, which prevents them from consuming any functional unit resources. Note that store instructions actually access the cache when they commit.
- For imprecise exceptions, the branch target stored in the *Next_Block* field for the block is retrieved from the *BHB* and used as the next PC. This effectively skips the entire block of instructions.

If the input values stored in the *BHB* do not match those in the processor's current state, or if there is no entry for this block in the *BHB*, the processor core will take control and issue the instructions to the functional units for normal execution. The processor core will continue to update the *BHB* whenever an instruction in a block commits.

3.2 Indirect Memory Referencing

In the load-store architectures, memory addresses only change if the corresponding input registers also change. Therefore, if the register inputs to a basic block differ, then the memory addresses calculated from these registers will also differ. Furthermore, recall that the *BHB* checks the contents of the data cache as well as the addresses being referenced. Consequently, even if the user program uses multiple levels of pointers, the *BHB* still detects the repetition of block inputs correctly.

3.3 Simulation Methodology

The *block history buffer* (BHB) can be implemented in various styles. Since our purpose is to illustrate the potential of a novel mechanism, we restrict our attention to evaluating only the proposed design, instead of comparing different design options. We use execution-driven simulations to investigate the performance potential that could be obtained by using the *BHB* to skip around the execution of all of the instructions in a basic block with repeating inputs. We modified the SimpleScalar Tool Set [8] for all of our experiments. The SimpleScalar processor has an extended MIPS-like instruction set architecture with modified versions of the *GCC* compiler (version 2.6.2), the *gas* assembler, and the *gld* loader.

The base superscalar processor used in this study contains 4 integer ALUs, 1 integer multiply/divide unit, 4 floating-point adders, and one floating-point multiply/divide unit. It can issue and commit up to four instructions per cycle with dynamic instruction reordering. The execution pipeline, the branch prediction unit, and a two-level cache are simulated in detail. All programs are compiled with the *-O2* optimization level using SimpleScalar's *GCC* compiler. The resulting programs are simulated on an SGI Challenge cluster with MIPS R10000 processors running version 6.2 of the IRIX operating system.

Three programs from SPEC92, *Alvinn, Ear*, and *Wordcount*, and 6 programs from SPEC95, *Compress, Go, Ijpeg, Li, M88Ksim* and *Perl*, were evaluated. The test input sizes were used for most of the programs. However, *Go* was driven with the *train* input size. *Word-count* used an input text file of 9871 lines, containing over 40,000 words.

3.4 Compiler Support

The *GCC* compiler saves all dead register information in the *REG_NOTE* field of its *RTX* structure in the flow analysis step. However, this information is inaccurate after it does register allocation. We added another flow analysis step right before assembly code is generated to obtain correct *REG_NOTEs*. Then we modified *GCC*'s assembly code generation step to encode dead register information in each instruction's annotation field [8]. The *block history buffer* can interpret the annotation field to identify the register number for each dead register output. While dead register outputs of a block are common, dead memory outputs are very rare. Consequently, we chose not to mark dead memory outputs at all so that all memory outputs are considered live at the end of a basic block.

3.5 Performance Results

To obtain a coarse upper bound on the performance benefit of the *block history buffer* mechanism, the simulations assumed that it takes one cycle to query the *BHB* and another cycle to update the registers and data cache. Also, each entry in the *BHB* can store any number of input and output values, but it is limited to 2048 entries with 2 read/write ports. The resulting speedup values shown in the last column of Table 13 are calculated by dividing the base execution time with the execution time obtained using the *BHB*. The resulting speedup values range from 1.01 to 1.37 with a typical value of 1.15. *Ear* has approximately 2% input locality and consequently, shows almost no speedup using the *BHB*. *Compress, Wordcount* and *Ijpeg* have good input locality but small skippable basic blocks. Hence, the speedup for these programs is relatively low (1.04 - 1.11). *Alvinn, Go, Li, M88Ksim*, and *Perl* have larger skippable basic blocks and large *task redundancy granularity* (Figure 4 and Table 12) which together produce speedup values for these programs between 1.15 and 1.37.

We next test the sensitivity of these speedup results to the number of fields available in each entry of the *BHB*. We choose to evaluate five cases based on the cumulative distributions of the number of block inputs and live outputs. For example, Figure 2 showed that 75 percent of all basic blocks have fewer than four register inputs, four live register outputs, three memory inputs, and two memory outputs. Thus, this configuration is used for the 75-percentile case. Similarly, 90 percent of all basic blocks have fewer than four register inputs, five live register outputs, four memory inputs and two memory outputs, and so forth. Table 14 shows the hardware configurations tested with the corresponding speedups listed in Table 13. Note that, since the hardware configurations for the 75, 80 and 85-percentile cases are the same, only three cases are actually compared here. We see that the performance improves gradually for all of the programs as the number of input and output values that can be stored increases. For the 95-percentile configuration, the speedup values are between 1.01 and 1.16 with a typical value of 1.10, which is close to the unlimited case.

Since each entry in the *BHB* records more than one register number and value pair, the time required to check the *BHB* and update the processor state may be longer than the 2 cycles assumed above. Table 15 shows the

Programs	4/4/3/2	4/5/4/2	5/6/4/3	Unlimited
Alvinn	1.04	1.05	1.05	1.15
Compress	1.06	1.06	1.06	1.07
Ear	1.01	1.01	1.01	1.01
Go	1.08	1.09	1.09	1.18
Ijpeg	1.01	1.08	1.08	1.11
Li	1.09	1.09	1.10	1.14
M88k	1.09	1.09	1.10	1.21
Perl	1.12	1.14	1.16	1.37
WC	1.04	1.04	1.04	1.04

Table 13: Speedup for different hardware settings as listed in Table 14

Percentile	Reg-In	Reg-Out	Mem-In	Mem-Out	Label
75	4	4	3	2	4/4/3/2
80	4	4	3	2	4/4/3/2
85	4	4	3	2	4/4/3/2
90	4	5	4	2	4/5/4/2
95	5	6	4	3	5/6/4/3
100	Unlimited	Unlimited	Unlimited	Unlimited	Unlimited

Table 14: Hardware settings to cover different basic block input and output requirements (in number of entries).

speedup obtained when varying the total time in cycles required to access the *BHB* and data cache, and to update the processor state. Here each entry in the BHB can hold 4 register inputs, 5 register outputs, 4 memory inputs, and 2 memory outputs, which corresponds to the 90-percentile case. We can see that the performance potential of the *BHB* is not particularly sensitive to the time required to interrogate the *BHB* and the data cache when a block is entered and to then update the processor state. For example, even if the delay takes 5 cycles, the speedup of block-reuse is still about 1.03 to 1.09, with a typical value of 1.06. This robust performance potential occurs because of the relatively large amount of time saved when a block's execution is skipped compared to the *BHB* overhead delays.

4 Related Work

Several techniques have been proposed to dynamically reuse values produced by instructions or to issue and execute portions of programs at a granularity coarser than a single instruction. These include dynamic instruction reuse [11], the block-structured instruction set architecture [15], the trace processor [2], and memory renaming [14].

Programs	2-cycle	3-cycle	4-cycle	5-cycle	6-cycle
Alvinn	1.05	1.04	1.04	1.04	1.03
Compress	1.06	1.04	1.04	1.03	1.03
Ear	1.01	1.01	1.01	1.01	1.01
Go	1.09	1.07	1.06	1.06	1.05
Ijpeg	1.08	1.08	1.07	1.06	1.06
Li	1.09	1.08	1.07	1.06	1.05
M88k	1.09	1.07	1.06	1.04	1.03
Perl	1.14	1.12	1.10	1.09	1.07
WC	1.04	1.03	1.02	1.01	1.01

Table 15: Potential speedup with different *block history buffer* delays for the 4/5/4/2 configuration.

Dynamic instruction reuse [11] stores the operands and output result of each instruction to eliminate the need to re-execute an instruction when its operands are the same as the last time the instruction was executed. This approach was introduced to make use of the squashed speculative execution of instructions due to branch mispredictions. Three reuse schemes were evaluated - 1) reuse based on operand values, 2) reuse based on operands' register numbers, and 3) reuse based on the register numbers and dependence chains. Reuse based on operand values was shown to be the most successful scheme. Our block reuse mechanism is essentially an extension of the instruction reuse approach to the basic block level, but our study did not save any values that were produced by speculation. Only committed instructions can update the *BHB*. Hence, our approach is conservative and could be enhanced with greater use of speculation.

Melvin and Patt [15] introduced a new instruction set architecture (ISA) based on basic blocks called the *block-structured ISA*. This ISA relies on the compiler to identify and merge basic blocks. All inter-block and intra-block data dependences are determined by the compiler and marked as such in the instructions. All instructions in a basic block are packed together to issue as a single unit. Only results that are live upon exit of a basic block update the general purpose registers. Register and memory writes are always delayed until the basic block is retired. Register reads always take the values currently in the register file upon entry to the block. This ISA supports speculative execution within basic blocks as well as across blocks. However, this ISA did not incorporate any block-level reuse as described in this paper.

The trace processor [2] uses a *trace cache* to store the instructions in a trace. A trace does not necessarily end at a basic block boundary. The trace processor also uses hardware to detect dependences [13] dynamically and identify both registers that are used locally and those used across traces. The scope of this register identification is limited to the traces that have been dispatched to the trace processing units and are awaiting execution. Hence, the detection logic does not have to consider register outputs that are used after the currently executing traces. Our technique evaluated in this study, on the other hand, always uses the basic block as the fundamental unit for value prediction and reuse. The trace processor still executes every instruction in the program, while our block reuse approach skips all blocks whose execution is determined to be redundant.

Tyson and Austin [14] proposed dynamic memory renaming to better resolve the problem of issuing load instructions ahead of stores before the address calculation of a store is completed. Their approach uses a *value file* to record the values fetched and written by each load and store instruction respectively. This mechanism also identifies the relationship between each load/store pair via a *load/store cache*. When a load or a store instruction is executed, it reserves an entry in the load/store cache. A load queries the load/store cache for the store instruction that produces the result it will fetch. If the corresponding store instruction has completed execution, the value stored in the *value file* will be forwarded to the corresponding load. If the store instruction is still being executed, a functional unit ID will be returned. Since the predicted relationship between load and store pairs can be wrong, all load and store instructions still need be executed to verify the prediction. This scheme operates only on memory operations and functions at the instruction level. Our block reuse mechanism addresses both register and memory operations at the basic block level. Again, once the *block history buffer* identifies a redundant execution of a basic block, all instructions in the block are skipped. No verification is necessary.

5 Conclusion

Speculation and reuse have been shown to be successful in improving processor performance, while value prediction has been shown to make these two approaches even more successful. Current prediction and reuse approaches use the instruction as the base unit. In this paper, we have extended these ideas to the granularity of the basic block and found that they are still applicable.

Our experiments using a subset of the SPEC benchmarks and the SimpleScalar Tool Set show that basic blocks have varying degrees of predictable input and outputs values. The depth-1 input value locality of basic blocks ranges between 2.21 and 41.44 percent, while the depth-1 output value locality varies from 3.15 to 51.63 percent. We also find that 90% of the basic blocks have fewer than 4 register inputs, 5 live register outputs, 4 memory inputs and 2 memory outputs, while 4 register inputs, 4 live register outputs, 3 memory inputs and 2 memory outputs are sufficient to cover the requirements of 85% of the basic blocks.

The relatively high input and output value locality of basic blocks, as well as their limited numbers of inputs and outputs, provides the basis for our approach of applying reuse techniques at the basic block level. We proposed a hardware mechanism called the *block history buffer* to record the input and output values of basic blocks to thereby identify blocks with repetitive inputs. The execution of the instructions within basic blocks with repetitive upward-exposed inputs are redundant and can be skipped. We call this scheme *block reuse*. Simulation results showed that a 2048-entry *block history buffer* with enough input and output fields to cover the requirements of 90% of the basic blocks produced miss rates below 7%. Block reuse with this *block history buffer* can improve performance for the tested programs from 1% to 14% with an overall average improvement of 9% when using reasonable hardware assumptions.

We conclude that exploiting basic block input value locality by skipping over redundant computation in a basic block has the potential to produce moderate performance improvements in the types of programs tested in this study. Hardware implementation details are needed to determine if the actual cost of a mechanism such as our proposed *block history buffer* is commensurate with the performance realized. Our current and future work focuses on extending this result to exploit block-level value speculation with greater compiler support. We expect to see a larger performance improvement over the basic block reuse mechanism presented here when these more aggressive techniques are examined.

References

[1] A. Aho, R. Sethi, J. Ulman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] J. Smith and S. Vajapeyam. “Trace Processors:Moving to Fourth Generation Microarchitectures,”. In *IEEE Computer*, September 1997, volume 30, number 9, Page 68 - 74.

[3] J.-Y. Tsai and P.-C. Yew. “Performance Study of a Concurrent Multithreaded Processor”. In the *4th Int’l Symposium on High Performance Computer Architecture* (HPCA-4), Las Vegas, Feb., 1998.

[4] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, K. Chang. “The Case for a Single-Chip Multiprocessor”. In the *Proceedings of 7th Int’l Conf. on Architecture Support for Programming Languages and Operating Systems* (ASPLOS VII), 10/1996.

[5] M. Lipasti, C. Wilkerson and J. Shen. “Value Locality and Load Value Prediction”, In the *Proceedings of 7th Int’l Conf. on Architecture Support for Programming Languages and Operating Systems* (ASPLOS VII), 10/1996.

[6] K. Wang, and M. Franklin. “Highly Accurate Data Value Prediction using Hybrid Predictors”. In the *30th Annual Int’l Symposium on Microarchitecture* (MICRO’30), Dec. 1997.

[7] Y. Sazeides, J. Smith. “The Predictability of Data Values”. In the *30th Annual Int’l Symposium on Microarchitecture* (MICRO’30), Dec. 1997.

[8] *The Simplescalar Tool Set, Version 2.0*. Technical Report 1342, Computer Science Department, University of Wisconsin, Madison.

[9] M. Lipasti and J. Shen. “Superspeculative Microarchitecture for Beyond AD 2000”. In *IEEE Computer* September, 1997, volume 30, number 9, Page 59 - 66.

[10] M. Franklin. *Multiscalar Processors* Ph. D Thesis, University of Wisconsin, 1993.

[11] A. Sodani and G. Sohi. “Dynamic Instruction Reuse”. In the *24th Int’l Symposium on Computer Architecture* (ISCA), June, 1997.

[12] J. Steffan and T. Mowry. “The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization”. In the *4th Int’l Symposium on High Performance Computer Architecture* (HPCA-4), Las Vegas, Feb., 1998.

[13] S. Vajapeyam, T. Mitra. “Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences”. In the *24th Int’l Symposium on Computer Architecture* (ISCA), June, 1997.

[14] G. Tyson, T. Austin. “Improving the Accuracy and Performance of Memory Communication Through Renaming”. In the *Proceedings of the 30th Annual Int’l Symposium on Microarchitecture* (MICRO’30), December, 1997.

[15] S. Melvin and Y. Patt. “Enhancing Instruction Scheduling with a Block-Structured ISA”. In the *Int’l Journal of Parallel Programming*, Volume 23, Number 3, 1995. Page 221 - 243.

[16] A. Gonzalez and M. Valero. “Virtual Physical Registers”. In the *4th Int’l Symposium on High Performance Computer Architecture* (HPCA-4), Las Vegas, Feb., 1998.