Increasing the Instruction Fetch Rate via Block-Structured Instruction Set Architectures

Eric Hao, Po-Yung Chang, Marius Evers, and Yale N. Patt
Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122
ehao@eecs.umich.edu

Abstract

To exploit larger amounts of instruction level parallelism, processors are being built with wider issue widths and larger numbers of functional units. Instruction fetch rate must also be increased in order to effectively exploit the performance potential of such processors. Block-structured ISAs provide an effective means of increasing the instruction fetch rate. We define an optimization, called block enlargement, that can be applied to a block-structured ISA to increase the instruction fetch rate of a processor that implements that ISA. We have constructed a compiler that generates block-structured ISA code, and a simulator that models the execution of that code on a block-structured ISA processor. We show that for the SPECint95 benchmarks, the block-structured ISA processor executing enlarged atomic blocks outperforms a conventional ISA processor by 12% while using simpler microarchitectural mechanisms to support wide-issue and dynamic scheduling.

1. Introduction

To achieve higher levels of performance, processors are being built with wider issue widths and larger numbers of functional units. In the past ten years, instruction issue width has grown from one (MIPS R2000, Sun MicroSparc, Motorola 68020), to two (Intel Pentium, Alpha 21064) to four (MIPS R10000, Sun UltraSparc, Alpha 21164, PowerPC 604). This increase in issue width will continue as processors attempt to exploit even higher levels of instruction level parallelism. To effectively exploit the performance potential of such processors, instruction fetch rate must also be increased. Because the average basic block size for integer programs is four to five instructions, processors that aim to exploit higher levels of instruction level parallelism must be able to fetch multiple basic blocks each cycle.

Various approaches have been proposed for increasing instruction fetch rate from that of a single basic block per cycle. Some approaches [24, 1, 2, 20] extend the branch predictor and icache so that multiple branch predictions can be made each cycle and multiple, non-consecutive cache lines can be fetched each cycle. However, this extra hardware requires extra stages in the pipeline which will increase the branch misprediction penalty, decreasing performance. Other approaches [5, 8] statically predict the direction to be taken by a program's branches and then based on those predictions, use the compiler to arrange the blocks so that the multiple blocks to be fetched are always placed in consecutive cache lines. Although they eliminate the need for extra hardware, these approaches must rely on the branch predictions made by a static branch predictor which is usually significantly less accurate than those made by a dynamic branch predictor.

This paper presents a solution using block-structured ISAs that exploits the advantages of both compiler-based and hardware-based solutions by merging basic blocks together statically and providing support for dynamic branch prediction. Block-structured ISAs [14, 13, 22] are a new class of instruction set architectures that were designed to address the performance obstacles faced by processors attempting to exploit high levels of instruction level parallelism. The major distinguishing feature of a block-structured ISA is that it defines the architectural atomic unit (i.e. the instruction) to be a group of operations. These groups of operations are called atomic blocks. Each operation within the atomic block corresponds roughly to an instruction in a conventional ISA. This redefinition of the atomic unit enables the block-structured ISA to simplify many implementation issues for wide-issue processors.

Block-structured ISAs increase the instruction fetch rate of a processor through the use of an optimization called

1072-4451/96 $5.00 © 1996 IEEE

block enlargement. Block enlargement combines separate atomic blocks into a single atomic block, increasing the average size of the program's atomic blocks. By increasing the sizes of the atomic blocks, the instruction fetch rate of the processor is increased without having to fetch multiple blocks each cycle. Furthermore, the semantics of the block-structured ISA enable the processor to use a dynamic branch predictor to predict the successor for each block fetched. As a result, block-structured ISAs increase the instruction fetch rate without relying on extra hardware to fetch non-consecutive blocks out of the icache or foregoing the use of dynamic branch prediction.

In this paper, we define one instance of a block-structured ISA for a wide-issue, dynamically scheduled processor. We have constructed a compiler that generates block-structured ISA code, and a simulator that models the execution of that code on a processor with a real branch predictor and a real icache. We show that for the SPECint95 benchmarks, the block-structured ISA processor executing enlarged atomic blocks outperforms a conventional ISA processor by 12% while using simpler hardware to support wide-issue and dynamic scheduling.

This paper is organized into five sections. Section 2 gives an overview of block-structured ISAs, explaining how the block enlargement optimization works and how it increases instruction fetch rate. Section 3 discusses other approaches to increasing instruction fetch rate. Section 4 describes our block-structured ISA and the compiler and microarchitectural support needed to implement that ISA. Section 5 presents experimental results comparing the performance of our block-structured ISA to that of a conventional ISA. Concluding remarks are given in section 6.

2. Block-Structured ISAs

Block-structured ISAs [14, 13, 22] were designed to help solve the performance obstacles faced by wide-issue processors. Their major distinguishing feature is that the architectural atomic unit is defined to be a group of operations. These groups, known as atomic blocks, are specified by the compiler. When an atomic block is issued into the machine, either every operation in the block is executed or none of the operations in the block are executed. The semantics of the atomic block enable the block-structured ISA to explicitly represent the dependencies among the operations within a block and to list the operations within the block in any order without affecting the semantics of the block. These features simplify the implementation of a wide-issue processor by simplifying the logic required for recording architectural state, checking dependencies, accessing the register file, and routing operations to the appropriate reservation stations. By reducing hardware complexity, wide-issue implementations of a block-structured ISA will require fewer hardware resources than that of a wide-issue implementation of a conventional ISA, resulting in a faster cycle time or a shallower pipeline. In addition to these benefits, block-structured ISAs can increase the instruction fetch rate of a processor via the block enlargement optimization.

Block enlargement is a compiler optimization that increases the size of an atomic block by combining the block with its control flow successors. Figure 1 illustrates how block enlargement works. The control flow graph on the left consists of the atomic blocks A–E, each one ending with a branch that specifies its successor blocks. These branches are called trap operations to differentiate them from fault operations which will be described below. These blocks are analogous to the basic blocks in a control flow graph for a conventional ISA. The control flow graph on the right shows the result of combining atomic block B with its control flow successors C and D to form the enlarged atomic blocks BC and BD. Both blocks BC and BD are now control flow successors to block A.

To support the block enlargement optimization, a new class of branch operations, the fault operation, is included in block-structured ISAs. The fault operation takes a condition and a target. If the condition evaluates to false, the fault operation has no effect. If the condition evaluates to true, the execution of the atomic block to which it belongs is suppressed and the instruction stream is redirected to its target. When two blocks are combined, the trap operation at the end of the first block is converted into a fault operation. If a block is combined with its fall-through successor, then the condition of the resulting fault operation is the same as the original trap operation's condition. If a block is combined with the target of its trap operation, then the condition of the resulting fault operation is the complement of the original trap operation's condition. The target of the fault operation is the enlarged block that results from combining the first block with its other control flow successor. In figure 1, when blocks B and C are combined, the trap at the end of B is converted into a fault in block BC. The fault's condition is true whenever block D is suppose to follow block B in the dynamic instruction stream and the fault's target is BD. Block BD contains a corresponding fault operation with a complementary condition and a target that points back to BC.

The block enlargement optimization requires that the processor support speculative execution [1]. This is required because the operations which compute a fault operation's condition may be in the same block as the fault operation. For example, in figure 1, the operations which compute the condition for block B's trap operation may be in block B

[1] Satisfying this requirement does not necessarily require that the block-structured ISA processor include hardware not already found in a conventional ISA processor because any processor that uses dynamic branch prediction must also support speculative execution, regardless of the ISA implemented by that processor.

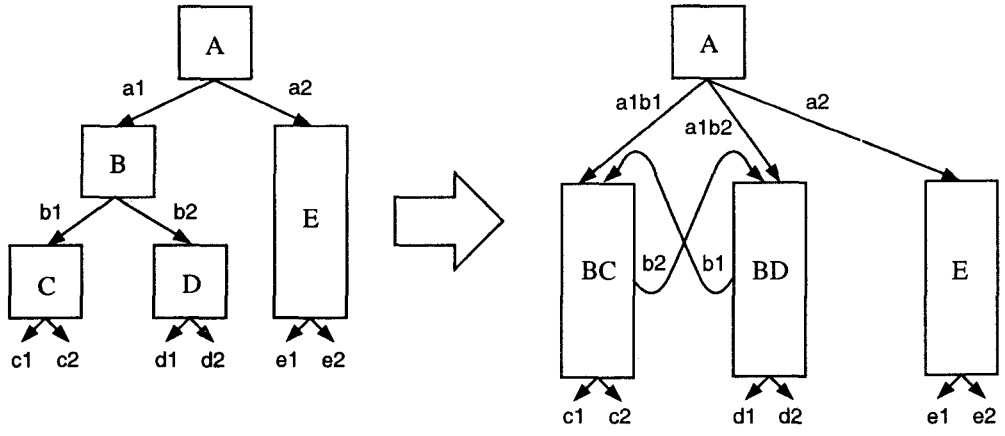

Figure 1. Combining atomic blocks into an enlarged atomic block.

itself. As a result, the operations which compute the conditions for block BC and block BD's fault operations are in block BC and block BD. To determine the correct successor to block A, the processor must speculatively execute either block BC or block BD and rely on the fault operations to correct the control flow if the speculation was incorrect. This assumes that block A's trap operation specifies that the $a1$ direction is to be taken.

Using the block enlargement optimization and the fault operation, block-structured ISAs are able to increase the instruction fetch rate without suffering the disadvantages associated with traditional approaches. By combining multiple basic blocks into a single, enlarged atomic block, the block enlargement optimization increases the instruction fetch rate without requiring the processor to fetch multiple non-consecutive cache lines. Furthermore, the use of fault operations enables the processor to use a dynamic branch predictor to choose which enlarged block is to be fetched next. However, for the block enlargement optimization to be effective, the optimization's effect on branch prediction accuracy and icache performance must be carefully considered. As more basic blocks are combined into an enlarged block, the probability that a fault operation within that block is mispredicted increases. Mispredicted fault operations incur an extra penalty not associated with ordinary branch mispredictions, because they cause all the work in their block to be discarded. Some of this work may have to be issued and executed again after the correct block is fetched. In addition, each time a block is combined with its successors, a separate copy of it is created for each successor. This duplication may increase the number of icache capacity misses during program execution and lower performance. This assumes that all the enlarged blocks formed from combining the block with its successors are accessed with sufficient frequency. If an enlarged block is never accessed, then it is never brought into the icache. The duplication incurred by such a block has no effect on the icache miss rate or the memory bandwidth used by the icache.

3. Related work

The majority of the approaches previously proposed for increasing the instruction fetch rate can be divided into two categories, compiler-based and hardware-based. The compiler-based schemes place the basic blocks to be fetched next to each other in the icache, eliminating the need for extra hardware. These schemes include trace and superblock scheduling [5, 8], predicated execution [7, 12, 18], and the VLIW multi-way jump mechanism [4, 10, 3, 15]. The hardware-based schemes extend the branch predictor and icache so that multiple branch predictions can be made each cycle and multiple non-consecutive cache lines can be fetched each cycle. They include the branch address cache [24], the collapsing buffer [1], the subgraph-level predictor [2], the multiple-block ahead branch predictor [20], and the trace cache [19].

Trace scheduling [5] and superblock scheduling [8] are compiler optimizations that enlarge the scope in which the compiler can schedule instructions. They use static branch prediction to determine the frequently executed program paths and place the basic blocks along these paths into consecutive locations, forming a superblock. The instructions within the superblock can then by optimized as if they were in a single basic block. The manner in which superblock scheduling combines blocks into superblocks is on the surface similar to that of the block enlargement optimization. The significant difference between the two approaches is that

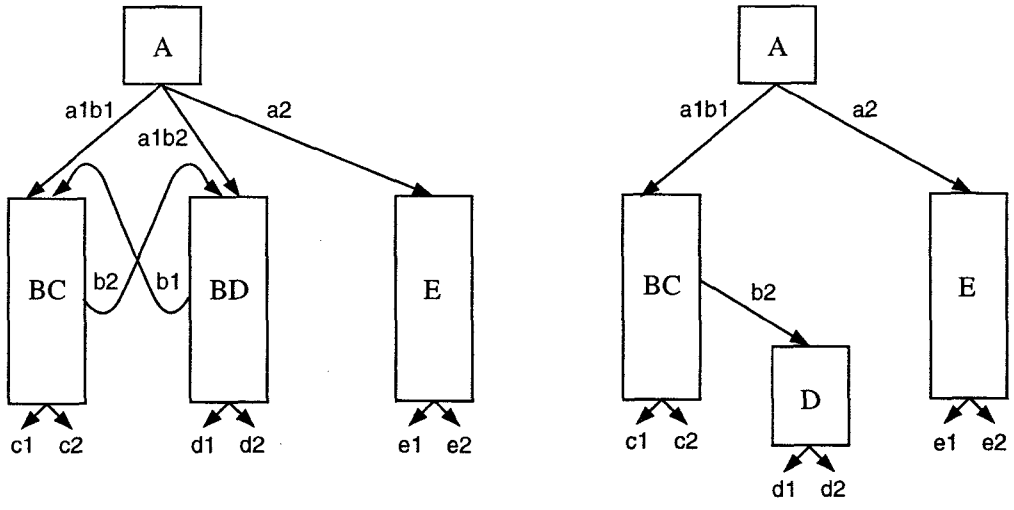

Figure 2. Using trace scheduling to combine basic blocks into a trace.

the block enlargement optimization uses dynamic branch prediction instead of static branch prediction to determine which basic blocks to fetch together. To illustrate this difference, figure 2 shows the results of applying block enlargement and superblock scheduling to the control flow graph from figure 1. The control flow graph on the left is the result of applying the block enlargement optimization and is identical to the one in figure 1. The control flow graph on the right is the result of applying superblock scheduling. The static branch predictor has predicted that block C is the most likely successor to block B so blocks B and C are combined to form a superblock. As a result, block B can only be fetched with block C. It can never be fetched with block D. In the block enlargement case, block B is combined with both of its successors to form enlarged atomic blocks BC and BD, allowing the dynamic predictor to choose the most likely combination. This extra degree of freedom provides a performance advantage for the block enlargement optimization over superblock scheduling because dynamic branch predictors usually achieve significantly higher prediction accuracies than static branch predictors.

Predicated execution [7, 12, 18] eliminates program branches by converting their control dependencies into data dependencies. Once a basic block's branch has been eliminated, it can be combined with its control flow successors to form a single basic block. Predicated execution has two disadvantages. First, it wastes fetch and issue bandwidth fetching and issuing instructions that are suppressed because their predicates evaluate to false. Second, by converting an instruction's control dependency into a data dependency, the program's critical paths may be lengthened. The processor must now wait for the new data dependency to be resolved instead of speculatively resolving the control dependency at fetch time. While predicated execution by itself may not be an effective mechanism for increasing instruction fetch rate, it can provide a significant performance benefit when used in conjunction with speculative execution [11] and other schemes for increasing fetch rate.

The VLIW multi-way jump mechanism [4, 10, 3, 15] combines multiple branches from multiple paths in the control flow graph into a single branch. Using this mechanism, basic blocks which form a rooted subgraph in the control flow graph can be combined into a single VLIW instruction. The branches for these basic blocks are combined into a single multi-way branch operation. This approach gives VLIW processors the means to fetch instructions from multiple basic blocks each cycle. However, because the operations within a VLIW instruction must be independent, it is critical that the compiler be able to find enough independent instructions to fill each VLIW instruction and be able to schedule the operations which evaluate the condition codes for the multi-way jump early enough so that the condition codes are available when the multi-way jump is issued. The compiler may have to delay the scheduling of certain operations in order to meet these requirements, lowering the instruction fetch rate of the processor.

The branch address cache [24], the collapsing buffer [1], the subgraph-level predictor [2], and the multiple-block ahead branch predictor [20] are hardware schemes that propose different ways to extend the dynamic branch predictor so that it can make multiple branch predictions each cycle. Because some of the branches may be predicted to be taken, these schemes all require the ability to fetch multiple non-consecutive lines from the icache each cycle. They all propose to meet this requirement by interleaving the icache. This general approach has two disadvantages. First, bank

conflicts will arise in the icache when fetching multiple lines from the same bank. To handle this conflict, the fetch for all but one of the conflicting lines must be delayed. This first disadvantage can be minimized if the icache is interleaved with a large enough number of banks. Second, because it is fetching multiple non-consecutive blocks from the icache, the processor must determine which instructions from the fetched cache lines correspond to the desired basic blocks and reorder the instructions so that they correspond to the order of those basic blocks. The processor will require at least one additional stage in the pipeline in order to accomplish these tasks. This additional stage will increase the branch misprediction penalty, decreasing overall performance.

The trace cache [19] is a hardware-based scheme that does not require fetching non-consecutive blocks from the icache. Its fetch unit consists of two parts, a core fetch unit and a trace cache. The core fetch unit fetches one basic block each cycle from the icache. The trace cache is a small cache that records sequences of basic blocks fetched by the core fetch unit, combining them into a single trace. If the branch predictor indicates that the sequence of basic blocks to be fetched matches a trace stored in the trace cache, then the processor is able to fetch multiple blocks that cycle by using the specified trace from the trace cache. If no matching trace is found, the processor is able to fetch only one basic block that cycle via the core fetch unit. As long as the processor is fetching its instructions from the trace cache, the trace cache is an effective means for fetching multiple basic blocks each cycle without incurring the costs associated with the other hardware-based approaches.

The trace cache and the block enlargement optimization are two very similar approaches to increasing instruction fetch rate. They both combine basic blocks into a single enlarged block (or trace) and use dynamic branch prediction to decide which enlarged block to fetch next. The key difference between them is that the trace cache combines its basic blocks at run-time while the block enlargement optimization combines its basic blocks at compile-time. By combining the blocks at run-time, the trace cache does not require changes to the instruction set architecture and does not increase the size of the executable. By combining the blocks at compile-time, the block enlargement optimization has the advantage of being able to use the entire icache to store its enlarged blocks instead of the small cache used in the trace cache approach.

Multiscalar processors [6, 21] are a new processing paradigm that does not fall into either the compiler-based or hardware-based categories. Multiscalar processors consist of a set of processing elements connected in a ring. Each processing element executes a task, a set of basic blocks specified by the compiler. The connecting logic among the processing elements forwards needed values along the ring and guarantees that the dependencies among the tasks are honored. Multiscalar processors eliminate the problem of fetching multiple cache lines from the icache each cycle by associating a L1 icache with each processing element. Each processing element accesses its own L1 icache for its task's instructions. As long as each icache achieves a sufficient hit rate, the multiscalar processor is able to fetch the equivalent work of multiple basic blocks each cycle. However, the multiscalar model raises new performance issues not found in traditional wide-issue processors. It is important that the compiler create tasks so that the work is evenly distributed and the communication among the tasks does not exceed the ring bandwidth.

4. Implementation details

4.1. The Block-Structured ISA Specification

We have defined a block-structured ISA that incorporates a subset of the features described in section 2. This ISA's architectural unit is the atomic block. The operations that can be found in an atomic block correspond to the instructions of a load/store architecture with the exception of conditional branches with direct targets. These branches are implemented as trap and fault operations. Each atomic block can contain any number of fault operations, but can contain at most one trap operation. Although an atomic block that ends in a trap operation may have more than two control flow successors, each trap operation specifies only two targets. The first target points to a block from the set of potential successor blocks given that the trap condition is true. The second target points to a block from the set of potential successor blocks given that the trap condition is false. Each trap operation also specifies the log of the total number of control flow successors for the trap's atomic block. Section 4.3 describes how the dynamic branch predictor uses this information to accurately predict the control flow successor for each atomic block.

4.2. The Block-Structured ISA Compiler

We implemented a compiler that is targeted for the block-structured ISA described above. This compiler is based on the Intel Reference C Compiler [9] with the back end appropriately retargeted. The Intel Reference C Compiler generates an intermediate representation of the program being compiled and applies the standard set of optimizations to that representation. We implemented a back end that takes this representation and applies a set of target-specific optimizations, allocates registers, and executes the block enlargement pass. During the block enlargement pass, the compiler attempts to combine as many different combinations of blocks as possible. The compiler begins with the first block of each function and continues until one of the

termination conditions listed below is met. The process is recursively repeated with the successor blocks of the newly formed enlarged block. The five termination conditions for the enlargement process are:

1. Atomic blocks can continue to expand until further expansion would cause the size of the enlarged block to exceed processor issue width. We restrict the maximum block size to the issue width in this block-structured ISA so as to avoid the complexity of supporting atomic blocks that require more than one cycle to issue. For our experiments the maximum block size will always be sixteen.

2. Each atomic block can contain at most two fault operations, which restricts the number of successor blocks for each block to at most eight. This restriction helps reduce the size of the branch predictor (see section 4.3) without significantly reducing the fetch bandwidth used.

3. Blocks that are connected via a call, return, or indirect jump cannot be combined. Mechanisms to support multiple successor candidates for such operations have not yet been developed.

4. Separate loop iterations are not combined into enlarged blocks. This restriction helps reduce the code expansion due to block enlargement without significantly affecting performance.

5. Blocks in library functions are not combined. Currently, we do not have the source code for our system library functions so we are not able to recompile them with the block enlargement optimization.

4.3. The Block-Structured ISA Processor

The block-structured ISA processor modeled in our experiments is a sixteen-wide issue, dynamically scheduled processor that implements the HPS execution model [16, 17]. The processor supports speculative execution as required by block-structured ISAs (see section 2). It can fetch and issue one atomic block each cycle. Each atomic block can contain up to sixteen operations. Dynamic register renaming removes any anti and output dependencies in the dynamic instruction stream. The processor can hold up to 32 atomic blocks at a time, equaling a maximum of 512 operations. As soon as its operands are all ready, an issued operation is scheduled for execution on one of sixteen uniform functional units whose latencies are listed in table 1. The processor has a 16KB L1 dcache and a perfect L2 dcache with a six cycle access time. The size of the processor's L1 icache is varied in our experiments, but the L2 icache is always modeled as perfect with a six cycle access time.

Instruction Class	Exec. Lat.	Description
Integer	1	INT add, sub and logic OPs
FP Add	3	FP add, sub, and convert
FP/INT Mul	3	FP mul and INT mul
FP/INT Div	8	FP div and INT div
Load	2	Memory loads
Store	-	Memory stores
Bit Field	1	Shift, and bit testing
Branch	1	Control instructions

Table 1. Instruction classes and latencies

To predict the next atomic block to be fetched, the block-structured ISA processor uses a predictor based on the Two-Level Adaptive Branch Predictor [25]. Because each atomic block may contain multiple branches, the block-structured ISA predictor must be able to implicitly make multiple branch predictions each cycle. To do this, the Two-Level Adaptive Branch Predictor must be modified in three ways:

1. The size of each BTB entry must be increased so that it can store all the possible control flow successors for an atomic block.. For our simulations, this number will be eight. When the atomic block is first encountered, the two explicitly specified targets in its trap operation are stored in the BTB. The remaining six targets are filled in to the BTB as they are encountered due to fault mispredictions.

2. Because each block can have up to eight control flow successors, the predictor must now produce a three bit prediction instead of a one bit prediction to select the predicted successor block. To do this, the pattern history table (PHT) entry is modified to hold additional counters to predict the fault operations in addition to the normal two-bit counter which will now be used to predict the trap direction.

3. The branch history register (BHR) should be updated each cycle with a varying number of history bits, because the number of branches predicted each cycle varies. When predicting the control flow successor to an atomic block with only two control flow successors, the branch predictor is predicting the direction taken by exactly one branch and only one bit is needed to uniquely identify that prediction. However, if the BHR is updated with the entire three bit prediction value, then two potentially useful history bits will be unnecessarily shifted out of the BHR. To prevent this unnecessary loss of branch history, the

block-structured ISA predictor shifts in the minimum number of history bits required to uniquely identify the current prediction. This number is specified in the corresponding trap operation (see section 4.1) and stored in the BTB.

5. Experimental results

To evaluate the performance advantages of block-structured ISAs, we compared the performance of the implementation of our block-structured ISA described in section 4.3 to an identically configured implementation of a conventional ISA. The two implementations had the same number of functional units, cycle times, icache size, and dcache size. In reality, the reduction in hardware complexity provided by block-structured ISAs would have enabled the implementation for the block-structured ISA to either have had a faster cycle time or a shallower pipeline than that of the conventional ISA implementation. As a result, the performance gains measured were more conservative than they otherwise would have been.

The conventional ISA implemented was the load/store ISA that formed the basis of our block-structured ISA. This eliminated any architectural advantages the block-structured ISA may have had over the conventional ISA with the exception of those due to block-structuring. To generate the conventional ISA executables, we used a variant of the block-structured ISA compiler that was retargeted to the conventional ISA. This eliminated any unfair compiler advantages one ISA may have had over the other.

Table 2 lists the eight SPECint95 benchmarks used for our comparison [2], along with the input data sets, and the number of conventional ISA instructions required to run each benchmark to completion.

Figure 3 compares the performance of the block-structured ISA executables to the performance of the conventional ISA executables when using a 64KB, four-way set associative L1 icache. Although current day L1 icache sizes are on the order of 8–16KB, advances in device densities will make 64KB L1 icaches feasible for future processors. The graph shows the total number of cycles required to execute each benchmark. With the exception of the go benchmark, each block-structured ISA executable outperformed the corresponding conventional ISA executable. The block enlargement optimization reduced the execution time on average by 12.3%, with reductions that ranged from 7.2% for gcc to 19.9% for m88ksim. The go benchmark showed a 1.5% increase in execution time which was due to increased icache misses, as will be shown below.

[2] The SPECfp95 floating point benchmarks were omitted from the study because our work has focused on extracting instruction level parallelism from non-scientific programs that represent everyday applications, where the parallelism is irregularly distributed and harder to extract. We note that block-structured ISAs should be able to achieve even larger performance gains for the floating point benchmarks.

Benchmark	Input	# of Instructions
compress	test.in*	103,015,025
gcc	jump.i	154,450,036
go	2stone9.in*	125,637,006
ijpeg	specmun.ppm*	206,802,135
m88sim	dcrand.train	120,738,195
perl	scrabbl.pl*	78,148,849
vortex	vortex.big*	232,003,378
xlisp	train.lsp	187,727,922

Table 2. The SPECint95 benchmarks and their input data sets. * indicates the input set is an abbreviated version of the SPECint95 reference input set.

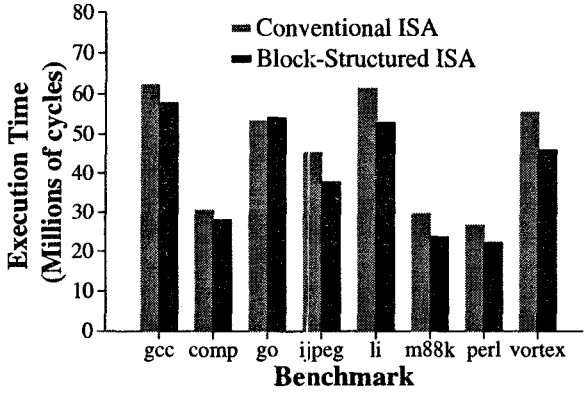

Figure 3. Performance comparison of block-structured ISA executables and conventional ISA executables.

To measure the performance impact of branch mispredictions on block-structured ISAs, figure 4 shows the same performance comparison as figure 3 except that perfect branch prediction is assumed. With perfect branch prediction, the average reduction in execution time is increased to 19.1%. Because both block-structured and conventional ISA executables incur about the same number of branch mispredictions, the increase in performance difference between the two ISAs indicates that branch mispredictions are more costly for block-structured ISAs. This is because good work must be removed from the machine for a fault misprediction and because the block-structured ISA is capable of fetching and issuing more work each cycle, increasing the performance loss for each cycle of branch misprediction penalty.

Figure 5 shows the average block size issued during exe-

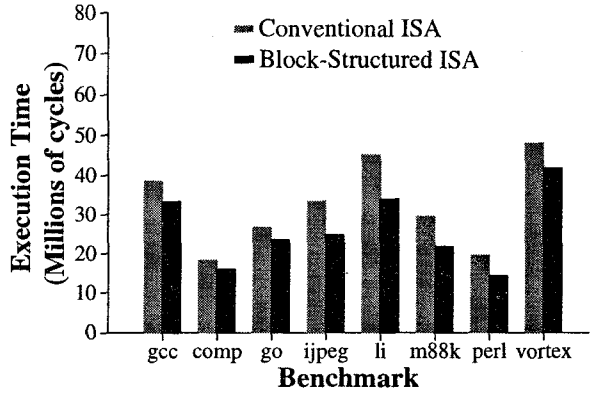

Figure 4. Performance comparison of block-structured ISA executables and conventional ISA executables while assuming perfect branch prediction.

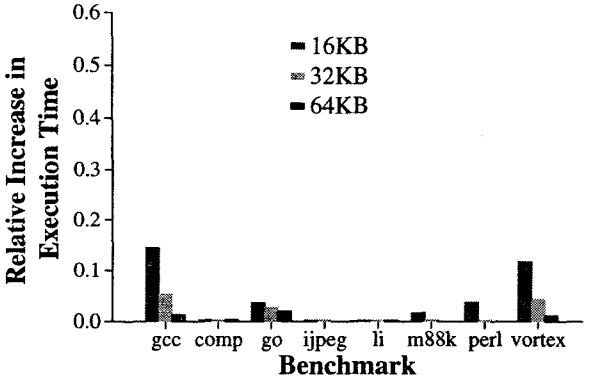

Figure 6. Relative increase in execution times for the conventional ISA executables over the execution time with a perfect icache.

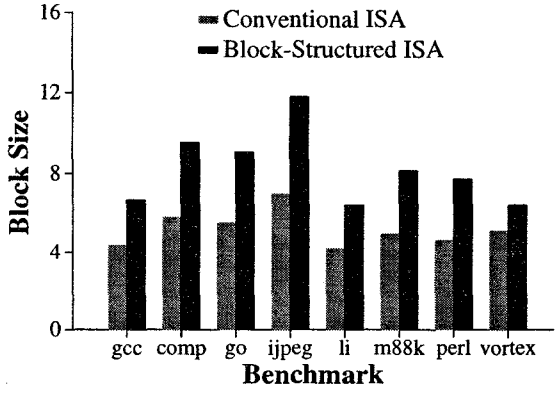

Figure 5. Average block sizes for block-structured and conventional ISA executables.

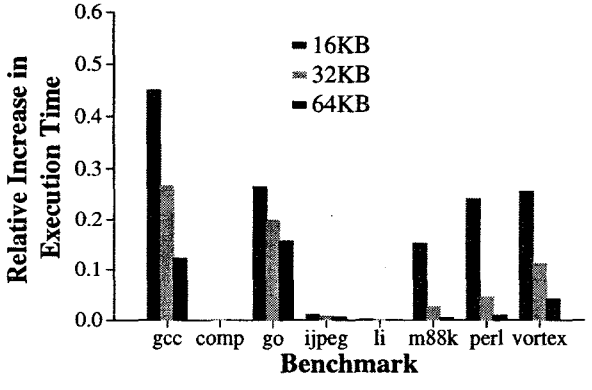

Figure 7. Relative increase in execution times for the block-structured ISA executables over the execution time with a perfect icache.

cution of each of the block-structured ISA and conventional ISA executables. Blocks that were not retired (i.e. that were issued after a branch misprediction) were not counted toward the average. Across all eight benchmarks, the average block sizes for the conventional ISA and block-structured ISA executables were 5.2 and 8.2 instructions. Despite increasing the block size by 58%, the block enlargement optimization still has left almost half the processors fetch bandwidth unused. The major reason why block enlargement was unable to create larger blocks was the occurrence of procedure calls and returns (see section 4.2). Their occurrence eliminated many opportunities for further block enlargement.

As discussed in section 2, by duplicating blocks, the block enlargement optimization may decrease the performance of the icache. To quantify this effect, we measured for each benchmark, the relative increase in execution time for a given icache size as compared to a perfect icache. This study did not model the icache effects due to operating system code. These effects may further decrease icache performance [23]. Figures 6 and 7 show the results of these comparisons for icache sizes from 16KB to 64KB. With the exception of the small benchmarks (compress, li, and ijpeg), the block-structured ISA executables show much larger performance decreases due to icache misses than the conventional ISA executables. The decrease is greatest in the gcc and go benchmarks which have many small basic blocks and many unbiased branches. As a result, by combining all the frequently executed combinations of basic blocks into enlarged blocks, the block-structured ISA executables for these benchmarks use significantly more icache space than the conventional ISA executables.

6. Conclusions

In this paper, we examined the performance benefits of using a block-structured ISA to increase the processor instruction fetch rate. Through the block enlargement optimization, the atomic blocks of the block-structured ISA could be combined to form larger blocks. As a result, larger units of work could be fetched from the icache each cycle without having to fetch non-consecutive cache lines. Furthermore, the block-structured ISA provides support for the use of dynamic branch prediction in selecting the successor block, rather than restricting the processor to static branch prediction.

We defined one instance of a block-structured ISA and implemented a compiler targeted to that ISA and a simulator to model the performance of a sixteen wide issue, dynamically scheduled processor that implements that ISA. Using this compiler and simulator, we compared the performance of programs executing on a block-structured ISA processor to the performance of programs executing on a conventional ISA processor for the SPECint95 benchmarks. The block-structured ISA processors achieved a 12% performance improvement over conventional ISA processors. This performance difference increased to 20% when perfect branch prediction was assumed. These performance improvements were due to the block enlargement optimization increasing the average block size from 5.2 instructions to 8.2 instructions. We also examined the block enlargement optimization's effect on icache performance. Because the block enlargement optimization duplicates blocks, the number of icache misses was larger for the block-structured ISA executables than for the conventional ISA executables. The difference was most significant for the gcc and go benchmarks.

The experimental results point to future directions in which the performance of processors implementing block-structured ISAs can be improved. Improving the branch prediction accuracy and icache hit rate and more fully utilizing the fetch bandwidth of the processor will further increase the performance difference between block-structured ISAs and conventional ISAs. Possibilities for achieving these goals include predicated execution, profiling, and inlining. Predicated execution can reduce the performance lost due to mispredicted branches by eliminating hard to predict branches. In addition, predicated execution can increase the fetch bandwidth used by eliminating branches that jump around small sections of the code. This optimization will create larger basic blocks which in turn will allow the block enlargement optimization to create even larger enlarged atomic blocks. Profiling can improve the icache hit rate by guiding the compiler's use of the block enlargement optimization. The amount of code duplication caused by the block enlargement optimization can be reduced if this optimization does not combine blocks that contain unbiased branches with their successors, thereby reducing the icache miss rate in exchange for smaller enlarged atomic blocks. Inlining can increase the fetch bandwidth used by eliminating procedure calls and returns, allowing the block enlargement optimization to combine blocks that previously could not be combined. In addition, using block-structured ISAs in conjunction with another fetch rate enhancing mechanism, such as the trace cache [19], may lead to even higher fetch rates without sacrificing icache performance.

We also plan to measure the performance gains that can be achieved by block-structured ISAs for scientific code. Those performance gains should be even greater than the gains achieved for the SPECint95 benchmarks because the branches that occur in scientific code are more predictable and the basic blocks are larger. These differences will reduce the performance penalty suffered by block-structured ISAs for branch mispredictions and increase the size of the blocks formed by the block enlargement optimization, resulting in even higher levels of performance.

This study focused on the improvements in instruction fetch rate provided by block-structured ISAs, ignoring the implementation benefits provided by block-structured ISAs. By reducing the hardware complexity of various microarchitectural mechanisms, block-structured ISAs enable the implementation of extremely wide issue processors. The performance benefits of these features would further increase the performance advantage of block-structured ISA processors over conventional ISA processors. In addition, as new algorithms are developed that increase the instruction level parallelism in programs that use them, the ability to implement wide issue processors will become even more important.

Acknowledgments

This paper is one result of our ongoing research in high performance computer implementation at the University of Michigan. The support of our industrial partners: Intel, Motorola, Hewlett-Packard, and NCR is greatly appreciated. We would also like to thank all the members of the HPS group for their contributions to this paper and the reviewers for their helpful suggestions.

References

[1] T. M. Conte, K. N. Menezes, P. M. Mills, and B. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *Proceedings of the 22st Annual International Symposium on Computer Architecture*, pages 333–344, 1995.

[2] S. Dutta and M. Franklin. Control flow prediction with tree-like subgraphs for superscalar processors. In *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 258–263, 1995.

[3] K. Ebcioğlu. Some design ideas for a VLIW architecture for sequential natured software. *Parallel Processing (Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing*, pages 3–21, Apr. 1988.

[4] J. A. Fisher. 2^n-way jump microinstruction hardware and an effective instruction binding method. In *Proceedings of the 13th Annual Microprogramming Workshop*, pages 64–75, 1980.

[5] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[6] M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 58–67, 1992.

[7] P. Hsu and E. Davidson. Highly concurrent scalar processing. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, 1986.

[8] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7(9-50), 1993.

[9] Intel Corporation. *Intel Reference C Compiler User's Guide for UNIX Systems*, 1993.

[10] K. Karplus and A. Nicolau. Efficient hardware for multi-way jumps and prefetches. In *Proceedings of the 18th Annual Microprogramming Workshop*, pages 11–18, 1985.

[11] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu. Characterizing the impact of predicated execution on branch prediction. In *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 217–227, 1994.

[12] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 45–54, 1992.

[13] S. Melvin and Y. Patt. Enhancing instruction scheduling with a block-structured ISA. *International Journal on Parallel Processing*, 23(3):221–243, 1995.

[14] S. Melvin and Y. N. Patt. Exploiting fine-grained parallelism through a combination of hardware and software techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 287–297, 1991.

[15] S.-M. Moon and K. Ebcioğlu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. In *Proceedings of the 25th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 55–71, 1992.

[16] Y. Patt, W. Hwu, and M. Shebanow. HPS, a new microarchitecture: Rationale and introduction. In *Proceedings of the 18th Annual Microprogramming Workshop*, pages 103–107, 1985.

[17] Y. N. Patt, S. W. Melvin, W. Hwu, and M. C. Shebanow. Critical issues regarding HPS, a high performance microarchitecture. In *Proceedings of the 18th Annual Microprogramming Workshop*, pages 109–116, 1985.

[18] D. N. Pnevmatikatos and G. S. Sohi. Guarded execution and dynamic branch prediction in dynamic ILP processors. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 120–129, 1994.

[19] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. Technical Report 1310, University of Wisconsin - Madison, Apr. 1996.

[20] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-block ahead branch predictors. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996. To appear.

[21] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22st Annual International Symposium on Computer Architecture*, 1995.

[22] E. Sprangle and Y. Patt. Facilitating superscalar processing via a combined static/dynamic register renaming scheme. In *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 143–147, 1994.

[23] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer. Instruction fetching: Coping with code bloat. In *Proceedings of the 22st Annual International Symposium on Computer Architecture*, pages 345–356, 1995.

[24] T.-Y. Yeh, D. Marr, and Y. N. Patt. Increasing the instruction fetch rate via multiple branch prediction and branch address cache. In *Proceedings of the International Conference on Supercomputing*, pages 67–76, 1993.

[25] T.-Y. Yeh and Y. N. Patt. Two-level adaptive branch prediction. In *Proceedings of the 24th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 51–61, 1991.