International Journal of Parallel Programming, Vol. 23, No. 3, 1995

# Enhancing Instruction Scheduling with a Block-Structured ISA

Stephen Melvin[1] and Yale Patt[2]

*Received July 8, 1993*

It is now generally recognized that not enough parallelism exists within the small basic blocks of most general purpose programs to satisfy high performance processors. Thus, a wide variety of techniques have been developed to exploit instruction level parallelism across basic block boundaries. In this paper we discuss some previous techniques along with their hardware and software requirements. Then we propose a new paradigm for an instruction set architecture (ISA): *block-structuring*. This new paradigm is presented, its hardware and software requirements are discussed and the results from a simulation study are presented. We show that a block-structured ISA utilizes both dynamic and compile-time mechanisms for exploiting instruction level parallelism and has significant performance advantages over a conventional ISA.

**KEY WORDS:** Instruction scheduling; instruction level parallelism; superscalar; VLIW; instruction set architecture.

## 1. INTRODUCTION

A basic block is defined by Aho *et al.*,[1] as a sequence of consecutive statements in which the flow of control enters at the beginning and leaves at the end. When basic blocks are small, performance suffers for several reasons. First, the instruction supply hardware is taxed. Changes in the flow of control expose the latency in prefetching and decoding from a new target address. In addition, small basic blocks limit multiple instruction per cycle execution. Issuing a large number of instructions in one cycle becomes impractical or at least very hardware intensive. Small basic blocks

[1] P.O. Box 2400, Berkeley, California 94702-0400. (E-mail: melvin@zytek.com).
[2] Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, Michigan 48109-2122. (E-mail: patt@eecs.umich.edu).



828/23/3-1

also complicate the task of the compiler in finding operations to overlap. As machines continue in the current trend of wider issue sizes, these are guaranteed to become more significant problems.

Exploiting instruction level parallelism across multiple basic blocks is straightforward in some instances. Many scientific programs have basic blocks that are fairly easy to enlarge. For example, when the bounds of a loop are known statically and there are no conditional tests inside, the loop can be trivially unrolled to provide a larger basic block. General purpose programs, however, generally fall into a different category. They often have small basic blocks and conditional branches that are hard to predict statically.

There has been a wide variety of techniques developed to exploit parallelism across multiple basic blocks. Early techniques involved global instruction scheduling by the compiler to move code between basic blocks. Currently there is a trend to supplement these techniques with architectural constructs (such as conditional instructions) and hardware mechanisms (such as speculative execution). There is also a trend toward the use of dynamic scheduling to further enhance the exploitation of instruction level parallelism. Dynamic scheduling is a microarchitectural mechanism that separates instruction *issue* from instruction *execution*. This mechanism has been implemented and proposed in many variations. The tag fowarding scheme of the IBM 360/91 originated the core idea behind dynamic scheduling.[2] HPS generalized the concept of tag fowarding to encompass all operations within a processor, including memory operations, and with enough backup state to allow dynamic branch prediction and precise exceptions.[3–5] Dynamic scheduling has particular advantages under variability in memory latency (e.g. cache hits vs. misses) and for dynamic memory disambiguation (e.g., when the compiler can't guarantee the independence of two memory references).

In this paper the idea of a block-structured instruction set architecture (ISA) will be introduced, which represents a logical extension of static and dynamic scheduling concepts. A block-structured ISA treats an entire group of instructions as an atomic unit, much as conventional ISAs treat individual instructions. This concept has several important implications that will be discussed. We will show that a block-structured ISA allows the compiler more flexibility in instruction scheduling, so that it can be more effective in uncovering global parallelism. Furthermore, the hardware can execute instructions in parallel more efficiently, and the instruction supply bottleneck is widened.

This paper is divided into six sections. In Section 2, we define some terms and provide some general background. Section 3 presents a survey of previous techniques for exploiting instruction level parallelism across multiple basic blocks. The hardware and software requirements of each are discussed along with their trade-offs. In Section 4 the concept of a block-structured ISA is detailed, and we discuss the hardware and software requirements. Section 5 presents some results from a preliminary simulation study of a block-structured ISA. We conclude with Section 6.



## 2. BACKGROUND

### 2.1. Atomic Blocks

As previously noted, a basic block is a sequence of consecutive instructions in which the flow of control enters at the beginning and leaves at the end. In this paper we define an *atomic block* similarly but as a *collection* of instructions rather than a *sequence* of instructions. There is no explicit sequencing of the instructions within an atomic block. Data flow dependencies within the atomic block are represented only by the source and destination fields of the constituent instructions, and there are no limitations on the order in which these instructions are stored in memory.

In an atomic block, only variables that are live upon exit from the block are assigned to architectural registers. A result not used outside the block is referenced by the index (relative to the beginning of the block) of the instruction generating the result. This index is converted to a physical register number at execution time.

Control dependencies between atomic blocks are represented through special instructions within the block known as *assert* instructions. Each assert instruction is associated with a target address. There may be many assert instructions (and thus many targets) within an atomic block. However, at run-time, only one target will eventually be resolved as the location of the next atomic block to execute. We will discuss atomic blocks and the use of assert instructions in Section 4.

### 2.2. Control Flow

Consider the various ways to alter the flow of control in a program. We classify control flow into four categories based on whether one-way or two-way branches are involved and whether or not targets are explicit or generated at run-time. Explicit targets are all addresses detectable statically from the instruction stream. Run-time targets include branches through registers, jump tables, computed branches and returns from subroutines. This branch categorization is shown in Table I. Note that a solid dot is used to represent an explicit target and an empty dot is used to indicate a run-time target. The symbols shown in Table I will be used later in the paper.

**Table I. Branch Types**

| | One-way branch | Two-way branch |
|---|---|---|
| Explicit target(s) | Unconditional branches<br>Calls to static targets | Conditional branches |
| Run-time target | Returns<br>Jumps through variables | Jump tables |

## 2.3. Block Entries and Exits

Basic blocks and atomic blocks are by definition single-entry, single-exit (**SESE**) constructs. There are other useful ways to classify units of work. Multiple-entry, single-exit (**MESE**) blocks are possible along with single-entry, multiple-exit (**SEME**) and multiple-entry, multiple-exit (**MEME**) blocks. The term *extended basic block* is used to refer to SEME-blocks (see Aho *et al.*[1]).

It is important to distinguish a block that has multiple targets as described earlier from a multiple-exit block. The latter applies only when the block has an *intermediate* exit point where not all instructions within the block are executed when that arc is traversed. Similarly, a multiple-entry block refers only to intermediate entry points, where not all instructions are executed when an intermediate entry arc is traversed.

All atomic blocks are SESE-blocks. An atomic block does not represent a sequence of instructions but merely a collection of work. Thus, it is impossible to jump into the middle or leave before all the work is done. However, we will see that an atomic block may have many possible successors.

## 2.4. Block Transformation

There are a variety of different ways to transform blocks to increase the available instruction level parallelism. Any two blocks joined by a one-way branch with an explicit address can be combined into a larger block. This may involve code expansion if there are other blocks that jump to the same location.

Another type of transformation involves code re-arrangement. Blocks with two-way branches with explicit targets can be arranged so as to create larger SEME-blocks without involving any changes to the contents of the individual blocks. This will be advantageous by not disrupting the sequential flow of instructions. Further advantages can be achieved in this case by performing optimizations on the entire SEME-block.

Many scheduling techniques involve moving instructions between two blocks across an arc. In the case that the arc is one side of a two-way branch, special care must be taken. An instruction following a two-way branch may be moved past the branch as long as any side-effects of this instruction can be undone in the case that the other direction of the branch is taken. This may involve inserting *fix-up code* in the other path or the use of special backup hardware.

We will discuss these block transformation techniques as well as others in more detail in the next section. For now, note that in the case of atomic blocks, it is possible in principle to move any instruction across any explicit arc. The combination of two or more blocks to create an enlarged atomic block involves a complete re-optimization of the resulting atomic block without the need to preserving architectural state for nonoptimized paths. This procedure is discussed in detail in Section 4.



## 3. PREVIOUS TECHNIQUES

There has been much previous work on exploiting instruction level parallelism across multiple basic blocks. Some early work was done in connection with VLIW machines. In these machines the compiler does all the scheduling of instructions and there are typically many concurrent operations. VLIW machines have been effectively applied only to scientific programs where the parallelism is relatively easy to detect. Basic blocks are large, branches are easier to predict statically and memory disambiguation is more straightforward.

Trace scheduling[6,7] is such a VLIW-based technique that involves optimizing for a particular path through a program. A particular trace is favored, and optimization is performed on that path. Instructions are moved across branches to fill in empty slots and maximize parallelism. The sequence of basic blocks along the trace represents a SEME-block that is being optimized.

Other forms of global scheduling, such as percolation scheduling,[8] involve more general approaches that do not require a particular favored trace through the program. However, like trace scheduling the state of the machine must be preserved at the intermediate exit points. Thus, fix-up code may have to be inserted. For example, consider the program structure example illustrated in Fig. 1. Each box in this figure represents a basic block in the original program. We will refer to this sample structure

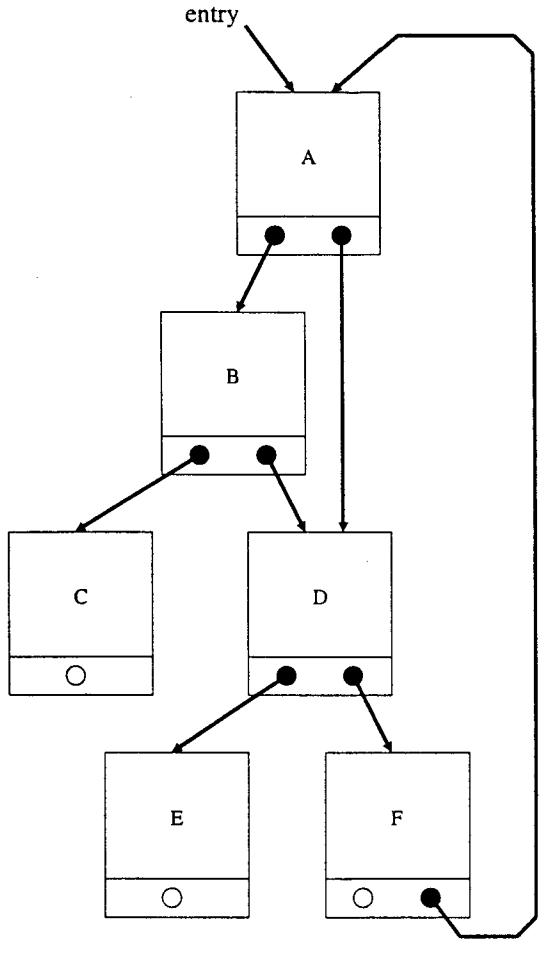



Fig. 1. A program structure that represents a basic block in the original program.

throughout the rest of the paper. If the basic block labeled $F$ is optimized with $D$, instructions may be moved up from $F$ to $D$. However, it may be necessary to insert instructions into $E$ in order to reverse the effect of $F$'s instructions in the case that the branch from $D$ to $E$ is taken. It may also be necessary to create additional instructions in $D$ itself in order to preserve the necessary information.

Today, superscalar processors are being developed with increasing numbers of function units. Many of these processors rely on the run-time



detection of independent instructions to be executed in parallel. Some also have dynamic scheduling in which instructions are executed out-of-order according to when their operands are available. However, these hardware constructs don't obviate global instruction scheduling. Some recent work in global instruction scheduling for superscalar processors is reported by Bernstein and Rodeh[9] and Chang *et al.*[10]

Software pipelining is another technique that has been widely used. It was used at least as early as the CDC 7600 and has been used extensively in VLIW machines. Some more recent work applies software pipelining to superscalar machines.[11,12] It involves carefully scheduling instructions statically to allow work from multiple iterations of a loop to overlap. This approach is most effective for loops with particular types of control structures. Its application to general purpose programs with frequent branches and irregular loops is difficult at best.

Hardware support for conditional and speculative execution can remove some restrictions on how instructions are scheduled across basic block boundaries. Under conditional execution, the result of an operation is retained or discarded based on a run-time condition. In its simplest form, conditional execution doesn't require any backup hardware because there is nothing that needs to be undone. For example, a conditional move instruction (such as in the Alpha and Sparc V9 architectures) will either store a value to a register or not store that value, based on a second value stored in another register.

Conditional moves can be used to eliminate branches in some cases. For example, suppose the basic block $B$ in Fig. 1 has only a single live variable upon exit. This variable could be updated conditionally based on the branch condition at the end of $A$. Then, $A$ and $B$ can be combined into a single basic block. There are several things to note about this type of optimization. First, if the branch condition for $A$ is known with a high degree of certainty statically, we would be better off optimizing for the expected path rather than using the conditional move. In particular, if $A$ almost always goes to $D$, then the compiler should not combine $A$ and $B$ into a single basic block. Also, if $B$ is large it may be better to wait for the branch test to be resolved before spending time on a computation that may be discarded. Thus, the decision to combine basic blocks using conditional moves involves a trade-off between the amount of conditional work, the probability of requiring the conditional work, and the relative cost of branches vs. conditional move instructions.

Some processors have implemented more general forms of conditional execution. Arbitrary instructions within a basic block can be made conditional based on a run-time condition. The resolution of this condition can be done before execution so that the instructions execute

without generating results if the condition is false. The Cydra-5 contained support for conditional or "predicated" instructions.[13] The VLIW tree instruction described by Aiker and Nicolau[14] and Moon and Ebcioglu[15] also support conditional execution. Conditional instructions consume machine resources even if the predicate is false. They may also add latency to the computation if the predicate is true. A project to compile to an architecture with conditional instructions is reported by Mahlke *et al.*[16] and Warter *et al.*[17] In these papers the term *hyperblock* is used to refer to an optimized SEME-block that contains conditional instructions.

Note that in the case that there is no backup mechanism, conditional execution optimizations must not prevent the recovery of the machine state if the predicate fails. Unused code can be removed from the optimized path, but values needed only in the case of intermediate exits can't be discarded. Also, predicated memory loads that have been promoted ahead of a branch must be tagged so that memory exceptions are delayed.

Other hardware models employ backup mechanisms to recover machine state in the case of the failure of a conditional test. This can allow even more flexibility in instruction scheduling. The "boosting" of instructions described by Smith *et al.*[18] employs this technique. In this case a shadow register set holds previous values until it is known that they are no longer needed. The necessity to fix up nonoptimized paths has been reduced, but boosting past multiple branches is difficult and the re-optimization process is restricted to what can be handled by the single backup register set.

More general support for speculative execution involves the implementation of multiple *checkpoints* and memory buffers that allow the machine to undo the effects of all operations, including memory writes.[19] Each checkpoint may require a backup register set, or it may be possible to save only the state of registers that have been changed.[20] A checkpoint/repair mechanism is typically used to support dynamic branch prediction, so that operations issued after a predicted branch can be discarded when a misprediction occurs. This type of mechanism is particularly relevant in a dynamically scheduled machine, where it is possible to issue instructions well in advance of when they are executed.

In all of the techniques discussed earlier, the conventional ISA poses limitations on how blocks of code can be optimized. Even if the hardware implements speculative execution with multiple checkpoints, the architecture remains a sequential model of execution. Branches are evaluated in a specified order, so at each intermediate exit point, recovery of the program state must be possible. In the next section we will discuss an alternative to this approach.



## 4. A NEW ISA PARADIGM

In the previous section we saw that global instruction scheduling by the compiler is now being applied to superscalar processors with conditional and speculative execution and dynamic scheduling. A block-structured ISA is a natural outgrowth of this combination. The concept of a block-structured ISA was presented by Melvin *et al.*[21] and further discussed and analyzed (see Ref. 22). In related work, Franklin and Sohi[23] discuss executing windows of instructions as single units.

In this section we introduce the concept of a block-structured ISA. This new paradigm is oriented around the execution of atomic blocks rather than individual instructions. It is important to note that we are describing an architectural concept, not a new implementation. This notion is critical to understanding how issue bandwidth can be increased. A block-structured ISA places requirements both on the hardware, which must have specific backup capabilities, and the software, which must use global scheduling techniques. In this section we describe such an ISA and discuss these software and hardware requirements.

### 4.1. The Block-Structured Concept

The fundamental idea behind a block-structured ISA is the specification of atomic blocks. Atomic blocks must appear to execute either completely or not at all. This means that backup to the beginning of the atomic block must be possible if every instruction can't complete. Thus, every instruction is in some sense speculative. Control flow exists at the atomic block level rather than at the instruction level. Each atomic block can have multiple target addresses, but at run-time only one of these targets will ultimately get resolved as the next logical atomic block to be executed. Target addresses can be explicit or dynamically generated.

Note that instructions within an atomic block can be stored in an arbitrary order. Flow dependencies are represented by the values of the instruction operands, but they don't restrict the placement of instructions within the atomic block. Values which are generated and used only within an atomic block are not stored in architecturally visible general purpose registers (GPRs), but in temporary physical registers. Only results that are live upon exit of the atomic block update the GPRs, and all reads from GPRs get the value stored upon entry to the atomic block.

An instruction which needs the result of an instruction in the same atomic block refers to it by its relative position. This is an intra-block index which will get converted into a physical register number an run-time. Note that by definition there are no anti or output dependencies within an

atomic block. There are no output dependencies since there will be at most one write to each GPR. There are no anti dependencies since any read from a GPR gets the original value of that GPR, even if it follows an instruction that writes to the same GPR.

Control flow between atomic blocks is handled through the use of assert instructions. Assert instructions are operations that test for a particular condition and either complete silently (generating no result) or produce an assertion signal. Each assert instruction is associated with a target address. When an assert instruction signals, the associated target address points to the next atomic block to execute. There may be multiple assert instructions within an atomic block and they are prioritized such that if multiple assert signals are generated, the target address associated with the highest priority signaling assert instruction is selected. The compiler guarantees that at least one assertion signal will always be generated.

There are two types of assert instructions: *faults* and *traps*. When a fault assertion signals, the atomic block containing that instruction must be undone. That is, the associated target address points to the atomic block to be executed after restoring the machine state to the condition it was in upon entry to the atomic block. When a trap assertion signals, the atomic block containing that instruction is not undone. The associated target address points to the next atomic block to be executed after the block containing the trap assert instruction is completed. Note that all trap assert instructions within an atomic block are mutually exclusive. That is, in the absence of signalling fault assertions, one and only one trap assertion will signal.

If the basic blocks $D$ and $F$ from Fig. 1 are combined into a single atomic block $DF$, the branch test at the end of $D$ would normally be converted to a fault assert instruction, while the branch test at the end of $F$ would be converted into a trap assert instruction. If the trap assertion signals, the results of block $DF$ can be retained. If, however, the fault assertion signals, then the block $DF$ must be discarded and execution must either continue with $D$, or with an atomic block $DE$ if the compiler has chosen to create one.

Alternatively, the compiler could choose to convert both branch tests into trap assert instructions. In this case, the signaling of the trap assertion from $D$ would cause transfer to block $E'$. The compiler would insert fix-up code as necessary into $E$ to create $E'$ to undo the effects that $F$ had on $D$. This kind of optimization is what would have been performed with a conventional ISA. We have the choice of putting fix-up code in $E$ and always completing the execution of $DF$, or backing out of $DF$ and redoing the work of $D$ in the case that the branch test in $D$ fails. Obviously it is desirable to minimize the amount of work that is discarded. This means that fault assertions must be used carefully, and their use must be weighed against the advantages of any increased enlargement of atomic blocks that is possible.



Conditional execution can also be incorporated into a block-structured ISA. For example, suppose as before that the basic block $B$ contains a single live variable upon exit. Further suppose that the compiler determines that the branch test for $A$ is randomly distributed between $B$ and $D$. Then, it would be reasonable to combine $A$, $B$ and $D$ into a single atomic block and use a conditional move instruction to select between the value computed in $B$ and the previous value for the variable in question.

An outline for a 64-bit block-structured ISA is shown in Fig. 4. In this example, we assume that register operands are 9 bits wide. If the type bit is zero, an architectural GPR is specified while if the type bit is one, the result of an instruction within the atomic block is referred to. This implies that there is a maximum atomic block size of 256 instructions. This also implies that any implementation of this architecture must have at least 256 physical registers to hold temporary results. Note that these registers might be distributed among reservation stations associated with each functional unit. If the implementation supports several atomic blocks executing concurrently, more than 256 registers might be desirable (although not necessary).

The target addresses are either explicit addresses or references to instructions within the atomic block (e.g. memory reads or computation instructions). The overall format of the atomic block has the header containing the target address list and the list of instructions. It would be possible to store the header separately from the rest of the atomic block. In this case we would need another pointer in the header to locate the list of instructions. Assert instructions are located in the body of the atomic block. There are no restrictions on where they may be located. Typically the compiler would put them as early as possible, after memory loads.

## 4.2. Compiler Support

A block-structured ISA has special implications for the compiler. Global instruction scheduling is just as important if not more important than in a conventional ISA. However, the mechanisms needed are different. Atomic blocks are not SEME-blocks, which is what conventional global instruction scheduling compilers optimize around. There are no intermediate exit points, and values internal to an atomic block are not allocated to GPRs. This makes some aspects of compiling simpler while others more complicated.

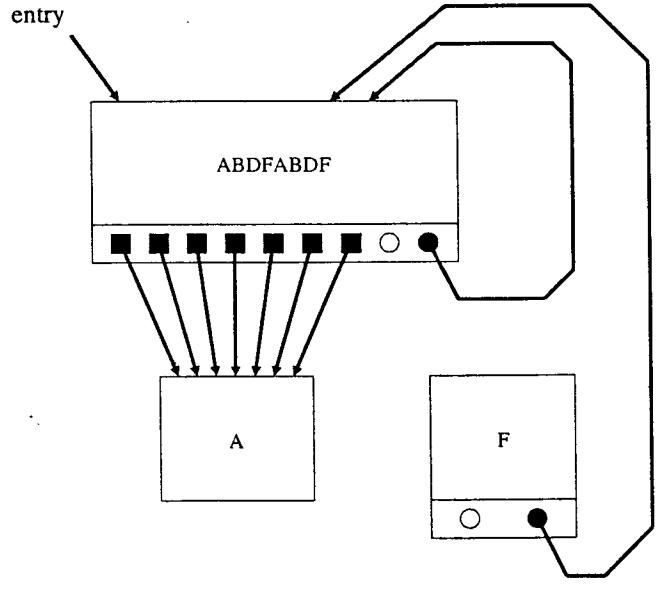



Fig. 2. An optimized structure in a block-structured ISA.

In creating atomic blocks, the compiler is responsible for choosing sequences of basic blocks to combine and creating assert instructions that verify the chosen paths. The manner in which this is done is non-trivial. The compiler has at its disposal traps and faults in addition to conditional instructions and prediction hints to the hardware.

Figure 2 illustrates how the structure in Fig. 1 might be optimized in a block-structured ISA. The basic blocks *A*, *B*, *D* and *F* have been combined and the loop back to *A* has been unrolled once. The new atomic block *ABDFABDF* represents the work of the eight constituent blocks re-optimized as a unit. The seven block exits that were created have been converted into fault assert instructions (represented by solid squares). Note that in some cases it may be possible to eliminate some of these intermediate tests. For example, it may be that the test associated with the second *F* block could only succeed if the test associated with the first *F* block also succeeded. In this case the first test can be eliminated.

For simplicity we show all the fault asserts pointing to block *A*. Depending on how the compiler evaluates the various intermediate branches, these fault asserts could point to other optimized atomic blocks. For example, if the first branch test from *B* fails, we could jump to an atomic block of *ABC*. Ideally we would like to minimize the number of fault assertions that occur dynamically. One way is to use conditional



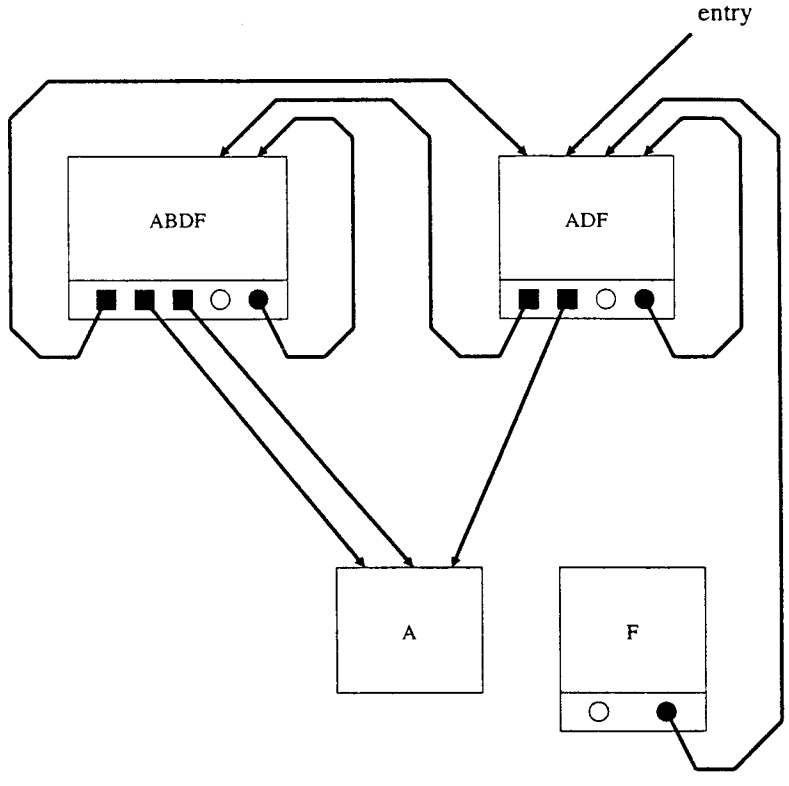


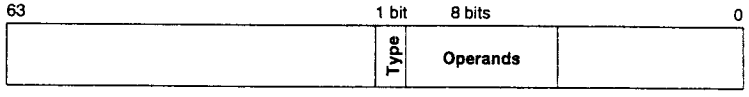

Fig. 3. Optimized blocks in the program structure.

move instructions to eliminate branch tests altogether. Another solution is to use fix-up code to convert faults into traps.

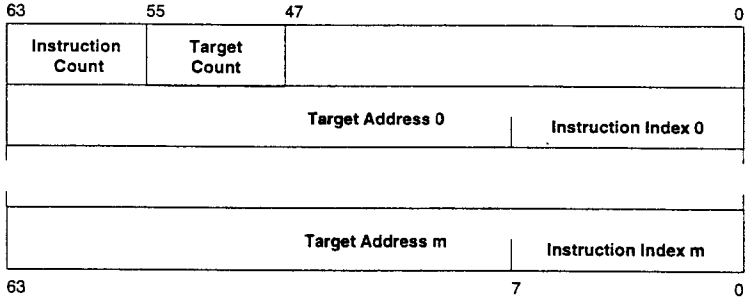
Another example of how the same program structure might be optimized is shown in Fig. 3. In this case we have not unrolled the loop but we have created optimized blocks for the case that *B* is executed and the case that it is excluded. This structure would be useful in cases where the patterns *ABDF* and *ADF* are both common and they tend to repeat themselves.

Enlarging atomic blocks must be done carefully. With each enlargement the chances of discarding the work increase. Thus, the decision of which arcs in the control flow graph of the program to enlarge across is complex. The run-time behavior of the specific arc in question as well as the characteristics of the blocks are critical to making this determination. The compiler would ideally use profiling information to make a better selection.

**Instruction:**




**Header:**




**Atomic Block:**

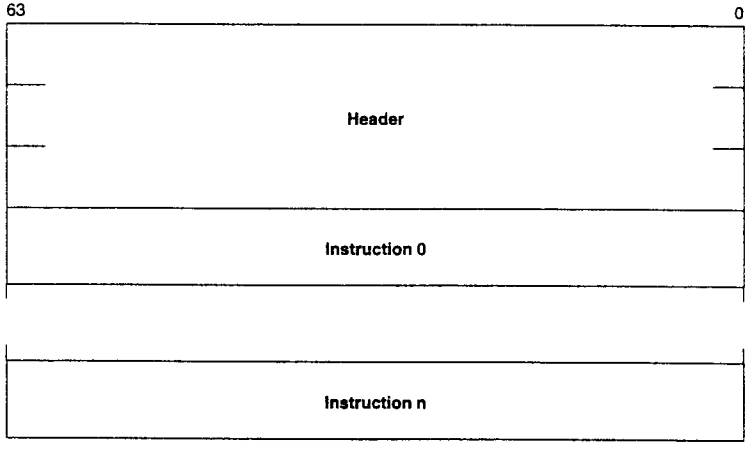



Fig. 4. An outline for a 64-bit block-structured ISA.



## 4.3. Hardware Support

Besides putting special requirements on the compiler, a block-structured ISA has specific hardware requirements as well. The atomic nature of the atomic block requires that there be at a minimum a backup register set and a memory write buffer. This allows register writes and memory writes that occur within an atomic block to be undone if necessary. There also must be a buffer to hold instruction results to be fowarded from one instruction to another within the atomic block. GPR writes are always held up until the atomic block is retired and GPR reads always take the value in the register upon entry to the atomic block.

The GPRs of the ISA represent logical registers while the processor internally operates on physical registers. As instructions are issued, each instruction that generates a result is allocated a physical register. The GPRs are also assigned to physical registers, and the GPRs for reading are assigned to different registers from the GPRs for writing.

The cost of this backup and register hardware is minimal in many cases because many of these mechanisms are already present in some implementations of conventional ISAs. Some kind of backup hardware is required for machines that support speculative execution. Also, the required register mechanism is already present on dynamically scheduled machines.

A processor implementing a block-structured ISA must have at least limited dynamically scheduling capability. This is mainly because the packing of instructions within an atomic block is not restricted. That is, an instruction that uses a value generated within an atomic block can be placed before the instruction that generates that value. One advantage of eliminating restrictions on instruction placement is that issue bandwidth is greatly increased. Suppose that an atomic block has very little inherent parallelism due to a long flow dependency chain. In a conventional ISA, this block would take a long time to issue. In a block-structured ISA, the block can be packed into the smallest possible number of cycles for issue. The flow dependencies will be enforced by the execution logic. This notion of *issue compression* eliminates the issue bottleneck associated with many dynamically scheduled machines.

A block-structured ISA processor supports speculative execution within the atomic block being executed since it must retain the ability to back up until execution of the entire atomic block is complete. It is also possible to support speculative execution between multiple atomic blocks. This would require backup capability for each active atomic block. The hardware would predict targets dynamically and continue to issue instructions along the predicted path. When multiple atomic blocks are executing,

signals from assert instructions can be processed to discard some ongoing work and proceed along another target while other work continues.

An important point to understand is that there is a trade-off between speculative execution exploited within the atomic block as a result of static analysis and speculative execution exploited across multiple atomic blocks as a result of dynamic analysis. In the case that the hardware can execute multiple atomic blocks simultaneously, we have the ability to discard each atomic block individually. If those atomic blocks were combined into a single atomic block by the compiler, they would only be able to execute as a unit.

The hardware can use dynamic informaton to predict the target address for the next atomic block. The compiler provides a static hint on which target to predict in the absence of dynamic information. Trap prediction and fault prediction require different handling. Target prediction in the case that an atomic block generates a trap assertion is fairly straightforward. The hardware can maintain an atomic block target buffer which holds the previous target(s) of the most recently executed atomic blocks. The use of history information and branch prediction counters can be used in much the same way as is common for conventional ISAs.

Fault prediction is more complex. In the simplest case, the hardware could assume that the compiler has been successful in generating the code such that faults are uncommon. It would thus be unnecessary to update any dynamic information when a fault occurs. Execution would merely continue with the atomic block pointed to by the fault instruction. Alternatively, the hardware could track faults and update prediction information when they occur. Note, however, that this involves updating the predicted target for the atomic block before the atomic block that contains the faulting assert instruction.

The ability to do fault prediction relies on the following property of the way code is generated. The atomic block pointed to by the target address associated with each fault instruction is in an equivalent class with the atomic block containing that instruction. If control is passed to a particular atomic block, it is OK to pass control instead to any other atomic block in the same equivalence class.

## 5. PRELIMINARY PERFORMANCE STUDY

In this section we report on a preliminary study in which a block-structured ISA was simulated. This study does not represent a comparison of block-structured versus conventional ISAs. Due to the complex nature of compiling to a block-structured ISA, a definite comparison of that sort will involve much long term research. The study does illustrate some experiences with generating code for a block structured ISA and compares the hardware and software speedup potential.



This study involved two basic components: a code generator and a run-time simulator. The code generator takes an intermediate representation of various UNIX utility programs and generates optimized code for a variety of different block-structured ISA configurations. The run-time simulator performs a cycle by cycle simulation of the program. System calls embedded in the original program are executed by the system on which the simulator runs. The simulator collects statistics on the execution of the entire program except for the system calls themselves, thus capturing the user level or unprivileged portion of the execution.

The optimization performed by the code generator was assisted by profiling information fed back from the simulator. The programs were run once using a first data set. Control flow statistics were collected for the entire program. The code generator then used this information to create enlarged atomic blocks. The resulting program was re-simulated using a different data set.

Atomic blocks with high frequency control transfer between them were combined into larger atomic blocks and re-optimized. In the case of loops, multiple iterations were unrolled. The run-time simulator also supports static information used to supplement the dynamic control flow prediction.

The parameters specifying the processor model fall into three categories: window size, issue model and enlargement. The *window size* is the total number of atomic blocks that may be active, partially issued but not fully executed, at any time. The issue model concerns the makeup of the instruction word, its width and breakdown between memory and ALU instructions. The third parameter concerns atomic block enlargement. The range of each parameter is shown in Table II.

We vary the window size in order to see how much parallelism is being exploited across atomic blocks that are predicted dynamically. The window size is specified in terms of the number of active atomic blocks allowed: 1, 4, or 256. If the window size is set to 1, this means that each atomic block is completely retired before the next atomic block can be issued. Thus, no inter-atomic block parallelism will be exploited.

The issue model specifies the format of the instruction word, that is how many instructions and what types can be issued in each cycle. Data we collected from the benchmarks indicate that the static ratio of ALU to memory instructions is about 2.5 to 1. Therefore, we have simulated machine configurations for both 2 to 1 and 3 to 1. We also simulated a model with a single memory instruction and a single ALU instruction since several commercial processors embody this format. Finally, we included a

**Table II. Simulation Parameters**

| Configuration parameter | Range of values |
|---|---|
| Window size | 1 Atomic block<br>4 Atomic blocks<br>256 Atomic blocks |
| Issue model | A. Sequential model<br>B. Instruction word = 1 memory, 1 ALU<br>C. Instruction word = 1 memory, 2 ALU<br>D. Instruction word = 1 memory, 3 ALU<br>E. Instruction word = 2 memory, 4 ALU<br>F. Instruction word = 2 memory, 6 ALU<br>G. Instruction word = 4 memory, 8 ALU<br>H. Instruction word = 4 memory, 12 ALU |
| Enlargement | Atomic blocks = original basic blocks<br>Atomic blocks enlarged |

*sequential* model, in which only a single instruction per cycle, of either type, is issued.

The third variable used in our study specifies whether or not atomic blocks are enlarged. Note that enlarged atomic blocks can take advantage of parallelism based on a static analysis of the program (in comparison to large windows, which can take advantage of parallelism based on a dynamic analysis of the control flow). The simulator implements a 2-bit counter for dynamic control flow prediction. The counter can optionally be supplemented by static control flow prediction information. This static information is used only the first time a control flow transfer is encountered; all future instances of the atomic block will use the counter as long as the information remains in the atomic block target buffer.

Several limitations of the dynamic control flow prediction scheme suggest that it may underestimate realistic performance. First, the 2-bit counter is a fairly simple scheme, even when supplemented with static control flow information. More sophisticated techniques yield better prediction accuracy.[24] Also, the simulator doesn't do *fault* assert prediction dynamically, only *trap* assert prediction. This means that control flow transfers to atomic blocks will always execute the initial atomic block first. A more sophisticated scheme would predict on faults such that repeated faults to the same atomic block would cause control flow transfers to jump directly to that atomic block.

For benchmarks we selected the following UNIX utilities. They represent the kinds of jobs that have been considered difficult to speed up with conventional ISAs.



- **sort** (sorts lines in a file)
- **grep** (print lines with a matching string)
- **diff** (find differences between two files)
- **cpp** (C pre-processor, macro expansion)
- **compress** (file compression)

The atomic block enlargement process employed is straightforward. The control flow arc densities from the first simulated run are sorted by use. Starting from the most heavily used, atomic blocks are enlarged until one of two criteria are met. The weight on the most common arc out of an atomic block can fall below a threshold or the ratio between the two arcs out of an atomic block can be below a threshold. Only two-way conditional control flow transfers to explicit destinations can be optimized and a maximum of 16 instances are created for each PC (this means that for example a loop will be unrolled at most 16 times). A more sophisticated enlargement procedure would consider correlations between branches and would employ more complex tests to determine where enlarged atomic blocks should be broken.

Each of the benchmarks were run under the conditions described earlier. Many statistics were gathered for each data point but the main datum of interest is the *average number of retired instructions per cycle*. This represents the total number of machine cycles divided into the total number of instructions which were retired (**not** executed). Un-retired, executed instructions are those that are scheduled but end up being thrown away due to assertion signals. The number of instructions retired is the same for a given benchmark on a given set of input data. Figure 5 summarizes the data from all the benchmarks as a function of the issue model. This graph represents data from each of the eight issue models. The six lines on this graph represent the three window sizes for single and enlarged atomic blocks.

One thing to note from this graph is that variation in performance among the different schemes is strongly dependent on the width of the instruction word and in particular on the number of memory instructions issued per cycle. In a case like issue model "*B*," where only one memory and one ALU instruction are issued per cycle, the variation in performance among all schemes is fairly low. However, for issue model "*H*," where up to 16 instructions can be issued per cycle the variation is quite large.

We see that atomic block enlargement has a significant performance benefit for all window sizes. In addition, there is significant parallelism that increased window sizes can take advantage of even for enlarged atomic blocks. Note that using enlarged atomic blocks with a window size of one

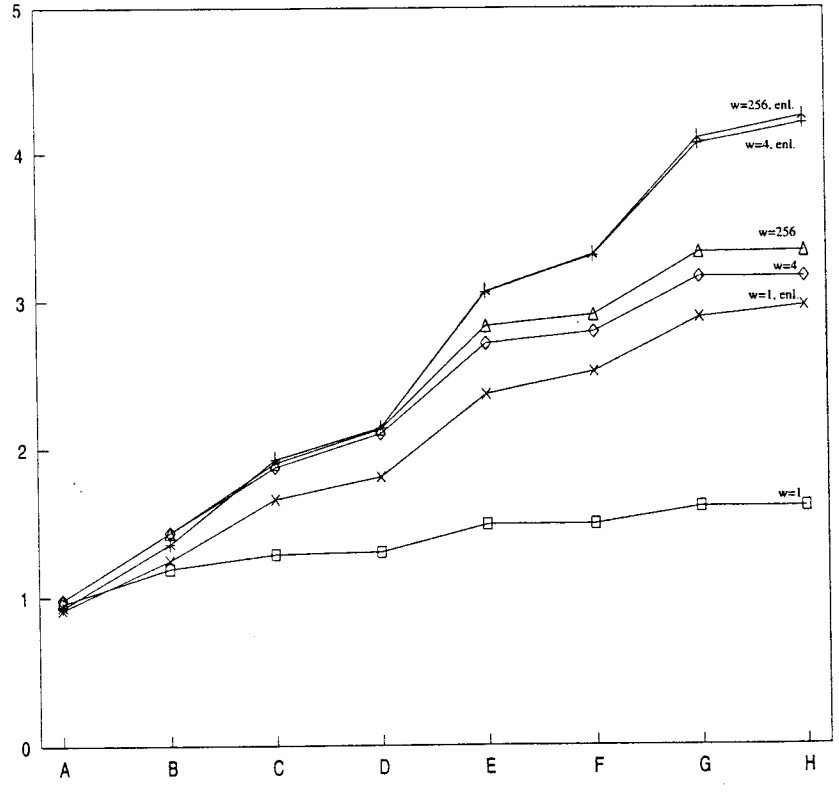



Fig. 5. A summary of the data from all the benchmarks as a function of the issue model.

doesn't perform as well as using single atomic blocks with a window size of four (although they are close). These are two different ways of exploiting speculative execution. In the case of enlarged atomic blocks without multiple checkpoints, the hardware can exploit parallelism within the atomic block but cannot overlap execution with other atomic blocks. In the case of a large instruction window composed of multiple unenlarged atomic blocks, we don't have the advantage of the static optimizations to reduce the number of instructions and raise utilization of issue bandwidth. Taking advantage of both mechanisms yields significantly higher performance than machines using either of the two individually can achieve.

This performance study represents only a beginning for the analysis of block-structured ISAs. We have only studied and optimized the code generation process. However, this study does suggest that there are some significant performance advantages to block-structured ISAs that warrant



further study. A more complete experiment would involve creating a complete optimizing compiler with global instruction scheduling optimizations as well as those studied here. We are continuing research into this area at the University of Michigan where a production compiler is being modified to target a block-structured ISA.

## 6. CONCLUSIONS

We have illustrated in this paper a new paradigm for an ISA that has important advantages for exploiting instruction level parallelism. A block-structured ISA is based on atomic blocks, which are collections of instructions that are treated as atomic units by the hardware. Control flow between atomic blocks is handled through the use of assert instructions. A block-structured ISA allows a larger unit of work to be optimized and increases the issue bandwidth of the machine.

The effective use of a block-structured ISA involves a combination of hardware and software mechanisms. The hardware must have dynamic scheduling as well as speculative execution capability. The compiler also has special requirements. It must create atomic blocks by combining work across branches in the original program. Re-optimization after the combination is a critical phase. The decisions of where to break the blocks and which arcs to combine across are a key trade-off.

The payoff for a block-structured ISA is a greatly enhanced potential for performance. The current trend of superscalar processors with wider issue widths is increasingly hampered by the one instruction at a time format of conventional ISAs. A block-structured ISA removes restrictions on how the compiler can combine work, since it is not necessary to preserve architectural state for nonoptimized paths. Multiple blocks can be combined in arbitrary ways to take advantage of the particular control flow paths of the program being executed.

A block-structured ISA also allows a multiple instruction per cycle processor to read in an entire collection of instructions without having to do any dependence checking between them. The compiler has already removed anti and output dependencies by labeling intermediate results with an intra-atomic block index. This decreases the amount of architectural state needed since named registers are only needed to convey results across atomic block boundaries.

An instruction format with the target addresses at the beginning of the atomic block can also lead to performance advantages. As soon as the first word of the atomic block is decoded, the instructions can be issued simultaneously with the pre-fetching of the target for the next atomic block. Thus, as the trend towards the combined use of static and dynamic scheduling continues, block-structured ISAs hold a future promise for high performance.

## REFERENCES


1. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques, and Tools*, Addison-Wesley (1986).
2. R. M. Tomasulo, An efficient algorithm for exploiting multiple arithmetic units, *IBM Journal of Research and Development* **11**:25–33 (1977).
3. Y. N. Patt, W. W. Hwu, and M. C. Shebanow, HPS, A new microarchitecture: rationale and introduction, *Proc., 18th Ann. Workshop on Microprogramming*, Asilomar, California (December 1985).
4. Y. N. Patt, M. C. Shebanow, W. Hwu, and S. W. Melvin, A C compiler for HPS I, a highly parallel excecution engine, *Proc., 19th Hawaii Int'l. Conf. on Sys. Sci.*, Honolulu, HI, January (1986).
5. W. W. Hwu and Y. N. Patt, HPSm, a high performance restricted data flow architecture having minimal functionality, *Proc., 13th Ann. Int'l. Symp. on Computer Architecture*, Tokyo (June 1986).
6. J. A. Fisher, Trace scheduling: a technique for global microcode compaction, *IEEE Trans. on Computers*, Vol. C-30, No. 7 (July 1981).
7. J. R. Ellis, Bulldog: a compiler for VLIW architectures, *The MIT Press* (1986).
8. A. Nicolau, Uniform parallelism exploitation in ordinary programs, *Proc. of the Int'l. Conf. on Parallel Processing* (August 1985).
9. D. Bernstein and M. Rodeh, Global instruction scheduling for superscalar machines, *Proc. Conf. Prog. Language Design and Implementation* (June 1991).
10. P. Chang, S. Mahlke, W. Chen, N. Warter, and W. Hwu, IMPACT: an architectural framework for multiple-instruction-issue processors, *Proc., 18th Ann. Int'l. Symp. on Computer Architecture* (May 1991).
11. M. Lam, Software pipelining: an effective scheduling technique for VLIW machines, *Proc. of SIGPLAN '88*, pp. 318–328 (June 1988).
12. R. Jones and V. Allen, Software pipelining: a comparison and improvement, *Proc. Micro-23*, pp. 46–56 (1990).
13. B. Rau, D. Yen, W. Yen, and R. Rowle, The Cydra 5 departmental supercomputer, *IEEE Computer*, pp. 12–35 (January 1989).
14. A. Aiken and A. Nicolau, A development environment for horizontal microcode, *IEEE Transactions on Software Engineering* (May 1988).
15. S.-M. Moon and K. Ebcioglu, An efficient resource-constrained global scheduling technique for superscalar and VLIW processors, *Proc., 25th Ann. Int'l. Symp. on Microarchitecture*, Portland (December 1992).
16. S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann, Effective compiler support for predicated execution using the hyperblock, *Proc., 25th Ann. Int'l. Symp. on Microarchitecture* (December 1992).
17. N. Warter, S. Mahlke, and W. Hwu, Reverse if-conversion, Technical Report, University of Illinois (June 1993).
18. M. Smith, M. Lam, and M. Horowitz, Boosting beyond static scheduling in a superscalar processor, *Proc., 17th Ann. Int'l. Symp. on Computer Architecture*, Seattle, Washington, pp. 344–353 (May 1990).
19. W. W. Hwu and Y. N. Patt, Checkpoint repair for out-of-order execution machines, *Proc. 14th Ann. Int'l. Symp. on Computer Architecture*, Pittsburgh, Pennsylvania (June 1987).
20. M. Butler and Y. Patt, A comparative performance evaluation of various state maintenance mechanisms, *Proc., 26th Ann. Int'l. Symp. on Microarchitecture*, Austin (December 1993).





21. S. W. Melvin, M. C. Shebanow, and Y. N. Patt, Hardware support for large atomic units in dynamically scheduled machines, *Proc., 21st Ann. Workshop on Microprogramming and Microarchitecture*, San Diego, California (November 1988).
22. S. Melvin and Y. Patt, Exploiting fine-grained parallelism through a combination of hardware and software techniques, *Proc., 18th Ann. Int'l. Symp. on Computer Architecture*, Toronto (May 1991).
23. M. Franklin and G. Sohi, The expandable split window paradigm for exploiting fine-grain parallelism, *Proc., 19th Ann. Int'l. Symp. on Computer Architecture*, Gold Coast (June 1992).
24. T.-Y. Yeh and Y. N. Patt, A comparison of dynamic branch predictors that use two levels of branch history, *Proc., 20th Ann. Int'l. Symp. on Computer Architecture*, San Diego (May 1993).