# Exploiting Fine-Grained Parallelism Through a Combination of Hardware and Software Techniques

Stephen Melvin

*Computer Science Division*
*University of California*
*Berkeley, California 94720*

Yale Patt

*Department of Electrical Engineering and Computer Science*
*University of Michigan*
*Ann Arbor, Michigan 48109-2122*

## *ABSTRACT*

It has been suggested that non-scientific code has very little parallelism not already exploited by existing processors. In this paper we show that contrary to this notion, there is actually a significant amount of unexploited parallelism in typical general purpose code. In order to exploit this parallelism, a combination of hardware and software techniques must be applied. We analyze three techniques: dynamic scheduling, speculative execution and basic block enlargement. We will show that indeed for narrow instruction words little is to be gained by applying these techniques. However, as the number of simultaneous operations increases, it becomes possible to achieve speedups of three to six on realistic processors.

## 1 Introduction

Advances that enhance performance can be broadly placed into two main categories: technological and architectural. Technological advances involve finding new materials and techniques to make gates that switch faster and memories that can be accessed faster. Architectural advances involve reorganizing these gates and memories to allow more operations to occur at the same time (i.e. a higher degree of overlap). Technological advances have dominated increases in speed in the past but the technology is approaching fundamental limits. Future increases in performance will be forced to rely more heavily on advances in computer architecture. In this paper, we focus on different architectural techniques applied to single instruction stream processors.

Several recent studies, for example [JoWa89] and [SmLH90], have suggested that most general purpose instruction streams have very little parallelism available, allowing a speedup of on the order of at most about two. This has caused some to say that intra-instruction stream parallelism is at its limit and that future increases in performance must rely solely on inter-instruction stream parallelism. We will show in this paper that the limits of single instruction stream performance are far from being reached. A significant amount of easily detectable parallelism actually exists in most general purpose instruction streams that is not exploited by existing processors.

The body of this paper has two sections. In section 2 we present three microarchitectural mechanisms: dynamic scheduling, speculative execution and basic block enlargement. These three closely related mechanisms are the focus of the performance enhancement techniques analyzed in this paper. Dynamic scheduling involves the decoupling of individual operations from others in the instruction stream, allowing them to be executed independently. Speculative execution exploits parallelism across branches and basic block enlargement is a technique used in conjunction with speculative execution and compile time effort. In section 3 a simulation study is described. We outline the simulation process, present the experimental methodology and reports on the results of the study.

## 2 Microarchitectural Mechanisms

In this section we present three microarchitectural mechanisms that are the subject of this paper: dynamic scheduling, speculative execution and basic block enlargement.

### 2.1 Dynamic Scheduling

Dynamic scheduling is a microarchitectural mechanism that allows the group of nodes that is currently active to be decoupled from each other. We use the term *node* to refer to an individual microoperation. Thus, the set of nodes that is issued together is not necessarily the same as the set that gets scheduled. By *issue* we mean the process of activating a node or preparing it for execution and *schedule* refers to the process of delivering a node to a function unit.

Dynamic scheduling mechanisms have been implemented and proposed in many variations. The tag forwarding scheme of the IBM 360/91 originated the core idea behind dynamic scheduling [Toma67]. Keller extended the idea and provided more background in [Kell75]. Register scoreboards, such as that of the CDC 6600, represent a limited form of dynamic scheduling. The HPS concept of *restricted data flow*, generalized the concept of tag forwarding to encompass all operations within a processor, including memory operations, and with enough backup state to allow dynamic branch prediction and precise exceptions. HPS was introduced in 1985 [PaHS85], [PMHS85] and is being reported on in continuing research [PSHM86], [HwPa86], [Hwu87].

One principle advantage of dynamic scheduling over static scheduling is that it allows individual nodes to be scheduled when they are ready to be executed, without holding up other nodes when they are not. In the presence of variable memory latency (i.e. cache misses) and the ability of the hardware to disambiguate memory addresses at run-time, this decoupling of nodes can lead to the discovery of valuable parallelism. Another main advantage of dynamic scheduling is that it allows parallelism across basic blocks to be exploited more efficiently. Static and dynamic scheduling are not mutually exclusive; many processors use some of both techniques.

In the case of static scheduling, the pipeline of the computer is exposed and the compiler fills in the node slots based on the static instruction stream. All memory operations are assumed to have a specific fixed latency (the cache hit latency) and there is typically a simple hardware interlock to stall the pipeline on a cache miss so that the forwarding of results within the pipeline works properly. The compiler preserves the original sequence of memory nodes in which there is a possible address match.

In the case of dynamic scheduling, there are no restrictions on the ordering of the nodes within the basic block as generated by the compiler. The machine issues all the nodes, fully decoupled from one another and any node result can be forwarded to any other node. Nodes are allowed to wait for memory or ALUs as long as necessary for their operands to become ready. The exact order of the execution of the nodes within the basic block is not determined. It may depend on memory cache contents and/or run-time operand values. Typically backup hardware is present to support speculative execution, allowing branch prediction misses to be processed efficiently. (However, speculative execution is an independent concept and can be applied to statically scheduled processors as well.)

There are several reasons which prevent statically scheduled machines from filling pipeline slots as effectively as dynamically scheduled machines. The first is variability in memory latency. When a memory read takes a cache miss and that value is needed by an operation, the statically scheduled machine must stall and wait for the data even if there are other operations which are available to be performed. It may be difficult for the compiler to organize the memory reads and ALU operations to prevent this from occurring. In machines with few resources this may not impose a significant performance penalty. Only if there is significant other work which can be done will dynamic scheduling pay off.

Another reason dynamically scheduled machines can fill pipeline slots more fully has to do with memory address disambiguation. Because the compiler only has static information available, it has to make a worst case assumption about matching memory addresses. Even if two memory nodes rarely or never point to the same location, the compiler is forced into imposing an artificial sequentiality unless it can determine no match will occur. An example of this is the case of array accesses with indices computed at run-time. Dynamically scheduled machines can do memory disambiguation at run-time and schedule the memory operations as appropriate to guarantee data flow dependencies are maintained. Some sort of dynamic memory disambiguation is usually present in all processors. In order to pipeline memory accesses, even if they occur in program order, it is necessary to have some checking logic to preserve data dependencies. This logic can then introduce stalls only when needed rather than the alternative which is to force no overlap of memory accesses. However, in the presence of out of order ALU operations and multiple ports to memory, a bigger advantage can be achieved by allowing memory accesses to occur out of order while still preserving flow dependencies.

Note that the ability of a statically scheduled machine to take full advantage of an execution pipeline depends to a large extent on the resources available. In the case that there is only one port to memory and one ALU, it is unlikely that there will be much difference in performance between static and dynamic scheduling.

## 2.2 Speculative Execution

Conventionally, *dynamic branch prediction* refers to a mechanism in which run-time information is used to predict the direction of branches. This is opposed to *static branch prediction* in which the compiler predicts branches using static information. The question over whether static or dynamic *information* is used to predict branches is secondary to performance. The real issue is whether or not **speculative execution** is supported, therefore we use this term rather than dynamic branch prediction.

In the case of speculative execution, a mechanism is present in which branches are predicted and operations are issued into the machine before the results of the prediction are confirmed (regardless of whether or not run-time information was used to predict the branch). In the absence of speculative execution, branches may be predicted before dynamic binding occurs. This might be the case in which branch prediction is used as a hint for the pre-fetcher to lower instruction fetch delay.

Speculative execution implies some form of backup capability. This is used in the case that a branch is predicted incorrectly. Note, however, that this backup logic may be very simple, it may consist of simply the ability to squash a write, flush the pipeline and re-fetch along another path. The manner in which speculative execution is able to increase performance depends on the extent to which the microarchitecture employs dynamic scheduling.

In the case of a purely statically scheduled machine without speculative execution, delay slots are usually implemented. These are used to overlap the pipeline delay of confirming a branch direction. Typically, the instruction immediately following a branch instruction is executed regardless of which direction is taken (in some cases the following **two** instructions). The compiler tries to fill the delay slot with work from the basic block being branched from or from one of the basic blocks being branched to. Ideally, the delay slot can be filled with work which must be performed regardless of branch direction. In some cases, however, it may only be possible to fill the delay slot with work that is only needed in one direction of the branch. Filling the delay slot this way sometimes requires the insertion of addition instructions in the alternate branch path to undo the effect of this work. This is called *fix-up code* and can be advantageous if the optimized branch direction is executed much more frequently than the alternate one. Finally, it may be necessary for the compiler to insert a NOP if no appropriate work can be found for the delay slot.

If speculative execution is added to a statically scheduled machine such as we have been describing, there are several advantages. First, delay slots can be filled more effectively. Operations may be placed into these locations that will get undone if the branch is taken a certain way. Also, this obviates fix-up code in the alternate basic block. In fact, moving operations up from a favored branch path could apply to slots within the basic block other than the branch delay slot. This may require more complex backup hardware however. In order to be able to backup an entire basic block, typically an alternate or backup register file is maintained and writes to memory are buffered in a *write buffer* before they are committed. We will show in the next section that with hardware such as this, basic blocks can be enlarged beyond their normal size to further increase pipeline slot utilization.

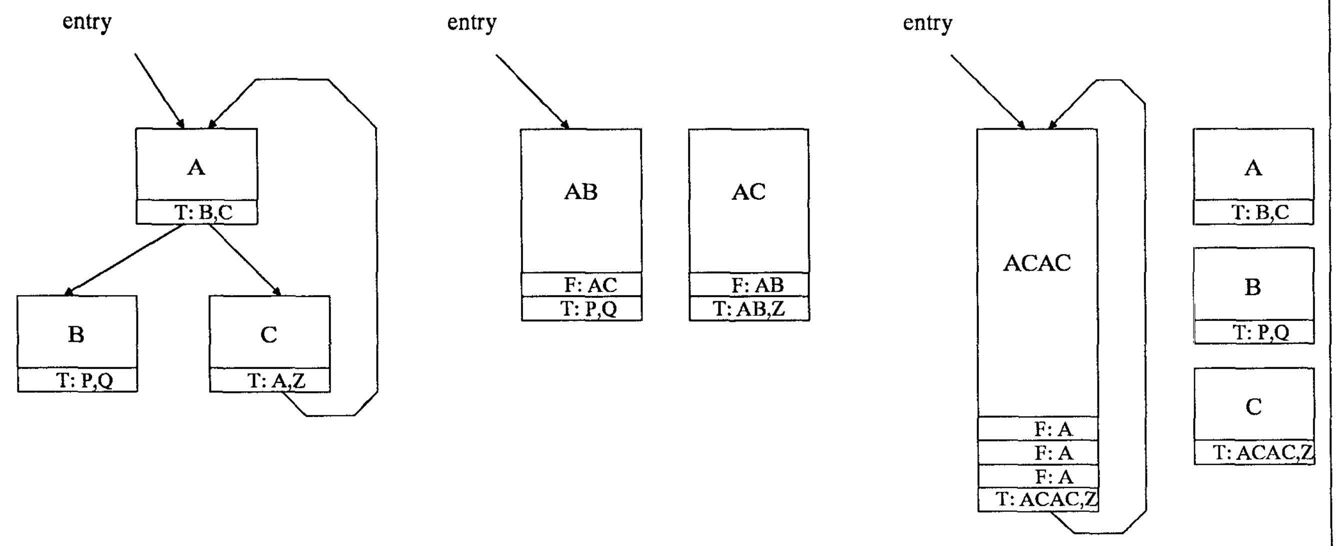
While statically scheduled machines can take advantage of speculative execution as we have just described, its use in a dynamically scheduled machine is a much more powerful concept. Here, the merging of operations continues across conditional branches and thus all work within several adjacent dynamic basic blocks contends for data path resources.

This use of speculative execution in conjunction with dynamic scheduling in this way introduces the concept of the *instruction window*. The instruction window represents a window of the dynamic instruction stream. The instruction window can be defined in several ways. We can count the number of active basic blocks, where an active basic block is one that has been partially or fully issued but not retired. We can also measure the window size by the number of operations in a particular state. For example, the number of active operations, the number of ready operations or the number of valid operations. An operation in the machine is always valid between the time it is issued and retired. It is active up until the time it is scheduled and it is ready only when it is active and schedulable.

## 2.3 Basic Block Enlargement

The basic block enlargement techniques we will describe in this section involve the exploitation of speculative execution to increase the size of the basic blocks as seen by the hardware. Suppose we have a statically scheduled processor with speculative execution and the ability to execute an entire basic block before committing the work. As we discussed in the previous section, this allows the compiler to move work into unused pipeline slots without having to

Figure 1
Basic Block Enlargement Examples



insert fix-up code in the alternate path. But the compiler need not stop with only a single basic block. Several basic blocks could be combined into a single unit, re-optimized and then generated as a single entity for execution.

In this case, at run-time this enlarged basic block would contain several embedded branch tests to confirm the branch prediction was correct. We call these operations *assert* nodes because they either execute silently or signal that some sort of backup is necessary. This is essentially what trace scheduling [Fish81] involves except in that case there is no speculative execution and no backup hardware, so the compiler must be more conservative. Loop unrolling and software pipelining are also examples of enlarging basic blocks in the absence of speculative execution.

Figure 1 illustrates two simple basic block enlargement examples. On the left is the original code where a basic block **A** branches to either **B** or **C** based on a run-time test. Basic block **C** further branches either back to **A** or to some other place. In the first example of basic block enlargement, shown in the middle of the figure, we create two new basic blocks, **AB** and **AC**, each of which has been re-optimized as a unit. At run time one of the two basic blocks, for example **AB**, would be issued. If the embedded branch test is incorrect, the entire basic block needs to be discarded and execution needs to continue with **AC**. The branch node which previously was part of **A** has been retained, but it has been converted from a *trap* node to a *fault* node and been given an explicit fault-to location.

A trap node is an operation which may signal that a prediction applied to a future basic block is incorrect, indicating that the basic block containing the trap node can be retained. When a fault node signals, on the other hand, the basic block containing the node must be discarded. Note that in the case that the **AC** basic block is never entered directly, the fault node could be eliminated. This is because the only way to get there would be the fault node in **AB**. Thus, there is no need to make the test that is guaranteed to succeed. However, if code which previously branched to **A** can potentially branch to **AC**, then the fault node needs to be kept.

This process of basic block enlargement can be continued recursively, allowing the creation of arbitrarily large basic blocks. Note that multiple iterations of a loop can be overlapped in this way, as shown on the right of figure 1. In this case, two iterations of the loop have been unrolled into a single basic block **ACAC**.

Consider the application of basic block enlargement to dynamically scheduled machines. A natural question which arises is why basic block enlargement is needed with dynamic scheduling. Since multiple basic blocks are issued into the machine, parallelism can already be exploited across conditional branches. Why then, is there an advantage to having the compiler (or possibly a hardware unit) create larger basic blocks? There are two reasons: larger scale optimization and issue bandwidth. First consider issue bandwidth. A problem that arises with basic blocks small relative to the number of operations that can be issued in a single cycle is that of low utilization of issue slots. The solution is to allow a larger unit of work to be packed into the set of nodes issued as a unit. In this way, the machine does not starve due to not being able to get enough work into it. In a machine which can issue 16 nodes in a single cycle running code with basic blocks that are are only 5-6 nodes large, this is obviously a critical technique.

The second reason for having basic block enlargement with dynamic scheduling has to do with the optimization of basic blocks. By combining two basic blocks across a branch into a single piece and then re-optimizing it as a unit, a more efficient basic block is achieved than would be created by just putting one after another. Suppose a compiler generates code like this:

```
        R0 <-- R0 + 4
        R1 <-- memory[R0]
        if (R1 > 0) jump to B else A
A:      R0 <-- R0 + 4
        R2 <-- memory[R0]
```

If the two basic block are combined across this branch, the artificial flow dependency through R0 can be eliminated. If the two basic blocks were issued together, this would be difficult or impossible to optimize.

Thus, dynamic scheduling needs basic block enlargement to support it. Now the opposite question arises: given basic block enlargement why do we need dynamic scheduling? The main reason is that basic block enlargement cannot take advantage of parallelism as flexibly as dynamic scheduling. As we have already seen, small basic blocks leads to problems with issue bandwidth and large scale optimizations. At the other extreme, a single enlarged basic block (as in the case of static scheduling) suffers from low efficiency. Each branch fault node within the enlarged basic block has a certain probability of signaling. For large basic blocks running non-scientific code, there is a point where efficiency will fall off to the point that enlargement is no longer worthwhile. The chances of having to discard a basic block are high enough that it doesn't pay to make it larger. The application of dynamic scheduling on top of basic block enlargement allows work with high probability of being needed (early in the window) to coexist with work with lower probability of being needed (late in the window). Thus, there is an optimal point between the enlargement of basic blocks and the use of dynamic scheduling.

## 3 Simulation Description and Results

In this section we describe the simulation study which was conducted and present the results from that study. Data was collected under a variety of machine configurations for a variety of benchmarks.

### 3.1 Simulator Overview

There are two basic components: the translating loader (tld) and the run time simulator (sim). The translating loader decompiles object code back into an intermediate form then does an optimized code generation for a specific machine configuration. The run time simulator reads in the translated code and does a cycle by cycle simulation of the program. System calls which are embedded in the original program are executed by the operating system on which the simulator is running. The run time simulator collects statistics on the execution of the entire program except for the the system calls themselves, thus it represents the user level or unprivileged portion of the execution.

Both the translating loader and the run-time simulator collect data which is gathered in a specified statistics file. The creation of a basic block enlargement file is handled by a separate program, which uses the statistics file. It combines basic blocks with high frequency branches between them into enlarged basic blocks. In the case of loops, multiple iterations are unrolled. The basic block enlargement file is then used as an input to the translating loader. The run-time simulator also supports static branch prediction (used to supplement the dynamic branch prediction) and a trace mode which allows the simulation of perfect branch prediction.

The division of the simulator into two pieces was done for convenience in data collection. The translating loader stores the programs fully decoded but it is not expected that an actual machine would be designed for such an instruction stream. For a small price in run-time decoding, the instruction bandwidth could be significantly reduced. Also, in the case of a dynamically scheduled processor, a more general format to be passed through a fill unit would obviate having the compiler know the exact function unit configuration.

The parameters which relate directly to the abstract processor model fall into four main categories: scheduling discipline, issue model, memory configuration and branch handling. The scheduling parameters concern static vs. dynamic scheduling and window size. The issue model concerns the makeup of the instruction word and the memory configuration parameters set the number of cycles for memory accesses and the cache size if any. The fourth parameter concerns basic block enlargement and branch prediction. The range of each parameter is shown below:

- **Scheduling Discipline**
  - Static Scheduling
  - Dynamic Scheduling, Window Size = 1 Basic Blocks
  - Dynamic Scheduling, Window Size = 4 Basic Blocks
  - Dynamic Scheduling, Window Size = 256 Basic Blocks
- **Issue Model**
  - **1.** Sequential Model
  - **2.** Instruction Word = 1 Memory Node, 1 ALU Node
  - **3.** Instruction Word = 1 Memory Node, 2 ALU Nodes
  - **4.** Instruction Word = 1 Memory Node, 3 ALU Nodes
  - **5.** Instruction Word = 2 Memory Nodes, 4 ALU Nodes
  - **6.** Instruction Word = 2 Memory Nodes, 6 ALU Nodes
  - **7.** Instruction Word = 4 Memory Nodes, 8 ALU Nodes
  - **8.** Instruction Word = 4 Memory Nodes, 12 ALU Nodes
- **Memory Configuration**
  - **A.** 1 Cycle Memory
  - **B.** 2 Cycle Memory
  - **C.** 3 Cycle Memory
  - **D.** 1 Cycle Cache Hit, 10 Cycle Miss, 1K Cache
  - **E.** 1 Cycle Cache Hit, 10 Cycle Miss, 16K Cache
  - **F.** 2 Cycle Cache Hit, 10 Cycle Miss, 1K Cache
  - **G.** 2 Cycle Cache Hit, 10 Cycle Miss, 16K Cache
- **Branch Handling**
  - Single Basic Block
  - Enlarged Basic Block
  - Perfect Prediction

The translating loader and the run-time simulator both do things differently depending on whether a statically scheduled or a dynamically scheduled machine has been specified. Thus, we need only to specify which of these two models is being used. In addition, we vary the window size in the case of dynamic scheduling. The window size can be specified in several ways. Here we specify it in terms of the number of active basic blocks allowed. We allow the window size to be 1, 4, and 256. If the window size is set to 1, this means that each basic block is completely retired before the next basic block can be issued. Thus, no inter-basic block parallelism will be exploited.

The issue model is the second main area in which the abstract processor model is parameterized. The issue model covers how many nodes and what types can be issued in each cycle, that is, the format of the instruction word. The data from the translating loader on the benchmarks we studied indicated that the static ratio of ALU to memory nodes was about 2.5 to one. Therefore, we have simulated machine configurations for both 2 to 1 and 3 to 1. We also simulate a model with a single memory node and a single ALU node. Finally, there is configuration we call the *sequential* model, in which only a single node per cycle is issued.

The third main area of parameterization of the abstract processor model is the memory configuration. The memory system is important to vary because we expect the tradeoffs for scheduling discipline to be different for different configurations. We considered memory access times of 1, 2 and 3 cycles. This represents the case for a perfect cache. Also, we simulate two different cache sizes, 1K and 16K bytes. The cache organization is two way set associative with a 16 byte block size. Note that the write buffer acts as a fully associative cache previous to this cache, so hit ratios are higher than might be expected. In all cases a cache miss takes 10 cycles.

The fourth variable used in the simulation study concerns how branches are handled. The run-time simulator implements a 2-bit counter branch predictor for dynamic branch prediction. The counter can optionally be supplemented by static branch prediction information. This information is used only the first time a branch is encountered, all future instances of the branch will use the counter as long as the information remains in the branch target buffer. Another option allows branches to be predicted 100%. This is accomplished by using a trace of basic blocks previously generated.

The main purpose behind the 100% prediction test is to establish an upper-limit on performance given that branch prediction doesn't constrain execution. These numbers shouldn't be taken to be realistic performance targets since perfect branch prediction is impossible. However, several limitations of the dynamic branch prediction scheme suggest that it may underestimate realistic performance. First, the 2-bit counter is a fairly simple scheme, even when supplemented with static branch information. It is possible that more sophisticated techniques could yield better prediction accuracy. Also, the simulator doesn't do branch *fault* prediction, only branch *trap* prediction. This means that branches to enlarged basic blocks will always execute the initial enlarged basic block first. A more sophisticated scheme would predict on faults such that repeated faults would cause branches to start with other basic blocks.

The benchmarks we have selected are UNIX utilities which represent the kinds of jobs that have been considered difficult to speed up with conventional architectures. The following is the list of benchmarks used:

- **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)

In order to handle basic block enlargement, two sets of input data were used for each benchmark. The first set was used in the single basic block mode and dynamic branch data was collected. Then, a basic block enlargement file was created using the branch data from the first simulation run. This file was used as input to the simulations for enlarged and perfect branch prediction studies. The input data used was different on these second simulations in order to prevent the branch data from being overly biased.

The basic block enlargement file creation employs a very simple procedure. The branch arc densities from the first simulated run are sorted by use. Starting from the most heavily used, basic blocks are enlarged until one of two criteria are met. The weight on the most common arc out of a basic block can fall below a threshold or the ratio between the two arcs out of a basic block can be below a threshold. Only two-way conditional branches to explicit destinations can be optimized and a maximum of 16 instances are created for original PC. A more sophisticated enlargement procedure would consider correlations between branches and would employ more complex tests to determine where enlarged basic blocks should be broken.

## 3.2 Simulation Results and Analysis

In this section we will present the simulation results from the benchmarks and the configurations discussed in the previous section. Each of the benchmarks were run under the conditions described above. There are 560 individual data points for each benchmark. This represents the product of the number of setting for each variable except that the 100% prediction test was only run on two of the scheduling disciplines (dynamic scheduling with window sizes of 4 and 256). Many statistics were gathered for each data point but the main datum of interest is the *average number of retired nodes per cycle*. This represents the total number of machine cycles divided into the total number of nodes which were retired (**not** executed). In the case of static scheduling, retired nodes and executed nodes are the same; in the case of dynamic scheduling, un-retired executed nodes are those that are scheduled but end up being thrown away due to branch prediction misses. Note that the number of nodes retired is the same for a given benchmark on a given set of input data.

Figure 3 summarizes the data from all the benchmarks as a function of the issue model and scheduling discipline. This graph represents data from the memory configuration 'A' for each of the eight issue models. That is, the data is for the case of a constant 1 cycle memory across a variety of instruction word widths. The ten lines on this graph represent the four scheduling disciplines for single and enlarged basic blocks and the two scheduling disciplines for perfect branch prediction case.

The most important thing to note from this graph is that variation is performance among the different schemes is strongly dependent on the width of the instruction word and in particular on the number of memory nodes issued per cycle. In a case like issue model '2', where only one memory and one ALU node are issued per cycle, the variation in performance among all schemes is fairly low. However, for issue model '8', where up to 16 nodes can be issued per cycle the variation is quite large.

We also see that basic block enlargement has a significant performance benefit for all scheduling disciplines. Implementing dynamic scheduling with a window size of one does little better than static scheduling while a window size of four comes close to a window size of 256. This effect is more pronounced for enlarged basic blocks than for single basic blocks. Also note that there is significant additional parallelism present that even a window size of 256 can't exploit. Thus, there is promise for better branch prediction and/or compiler techniques. It is interesting to note that using enlarged basic blocks with a window size of one still doesn't perform as well as using single basic 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 basic blocks without multiple checkpoints, the hardware can exploit parallelism within the basic block but cannot overlap execution with other basic blocks. In the other case of a large instruction window composed of single basic blocks, we don't have the advantage of the static optimizations to reduce the number of nodes 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.

Figure 4 summarizes the data as a function of memory configuration and scheduling discipline. This graph presents data for an issue model '8' for each of the seven memory configurations. Each of the lines on the graph represents one of the ten scheduling disciplines as in the previous graph (the first column in this graph is exactly the last column in the previous graph). Note the order of the memory configurations on the horizontal axis. The first three data points are for single cycle memory with various cache sizes, the second three points are two cycle memory and the last data point is three cycle memory.

Note that the slopes of the lines are all fairly close. What this means is that as a *percentage* of execution time, the lines higher on the graph are affected less by the slower memory than the lines lower on the graph. It might seem at first as though tripling the memory latency should have a much greater affect on performance than in any of these cases. Note that the memory system is fully pipelined. Thus, even with a latency of 3 cycles, a memory read can be issued every cycle to each read port. In the case of static scheduling, the compiler has organized the nodes to achieve an overlap of memory operations while in the case of dynamic scheduling, the scheduling logic performs the same function. This is particularly true for enlarged basic blocks where there is more of an opportunity to organize the nodes so as to hide the memory latency.

Even across all issue models there were no cases of steep curves as a function of memory configuration. This fact suggests that tolerance to memory latency is correlated with high performance. It is only machines that are tolerant of high memory latency to begin with which reach a high performance level with 1 cycle memory. Increasing the memory latency to 3 cycles has a minor affect on them. Machines that are intolerant of high memory latency don't perform well even with fast memory, so they have less to lose when the memory slows down. Being able to execute many nodes per cycle means having lots of parallelism available. If the memory slows down, this just means that there are more outstanding memory reads so more memory operations must be simultaneously executing. As we see from the graph, this is true even in the issue model '8' case, indicating that even more parallelism could be exploited with more paths to memory. Of course, the situation would be much different if the memory system were not fully pipelined. In that case a path to memory would have to go idle and we would expect a major decrease in performance as the memory slows down.

Figure 5 summarizes variations among the benchmarks as a function of a variety of machine configurations. We have chosen 14 composite configurations which slice diagonally through the 8 by 7 matrix of issue model and memory configuration. The five lines on the graph represent the performance on the given configuration for each of the benchmarks. The scheduling discipline is dynamic scheduling with a window size of four with enlarged basic blocks. As would be expected, the variation (percentage-wise) is higher for wide multinodewords. Several benchmarks take a dip in going from configuration '5B' to configuration '5D'. This is due to low memory locality; the 'B' configuration has constant 2 cycle memory and the

'D' configuration as 1 cycle hit, 10 cycle miss memory with a 1K cache. This effect is also noticeable in figure 4 in comparing the 'B' column to the 'D' column.

Figure 6 presents the operation redundancy for the eight issue models as a function of scheduling discipline. This graph illustrates one of the keys to speculative execution in particular and decoupled architectures in particular. Note that the order of the scheduling disciplines in figure 6 is exactly opposite from that in figure 3. Thus, the higher performing machines tend to throw away more operations. In the case of the dynamically scheduled machine with enlarged basic blocks and a window size of 256, nearly one out of every four nodes executed ends up being discarded. This is the price to pay for higher performance. However, note that this same configuration with a window size of four has significantly fewer discarded operations but the performance (from figure 5) is almost identical.

Now let's look into basic block enlargement in more detail. Figure 2 shows basic block size histograms for single and enlarged basic blocks averaged over all benchmarks. As would be expected, the original basic blocks are small and the distribution is highly skewed. Over half of all basic blocks executed are between 0 and 4 nodes. The use of enlargement makes the curve much flatter. However, a caveat should be made about this graph. The basic block enlargement techniques employed here should be distinguished from techniques that don't require speculative execution. In those cases, mainly successful with scientific code, basic blocks are enlarged by reorganizing the code and by inserting fix-up code if necessary. Then, the redundancy issue is still present, it is reflected in how many fix-up nodes are executed. Through the use of speculative execution, basic blocks could be made arbitrarily large. The performance would start to fall with the higher redundancy and the diminishing returns of large basic blocks.

## 4 Conclusions

Engineering involves the analysis of tradeoffs. In the case of computer engineering this analysis is particularly complex since decisions at one level may have wide reaching effects on other levels. Thus, an important facet of any serious computer design project is the careful consideration of how to allocate resources among the different levels of the machine. This fact has long been appreciated by most computer engineers. The following quote comes from 1946 in a discussion about which functions to include in an arithmetic unit by Burks, Goldstine and von Neumann:

> *"We wish to incorporate into the machine -- in the form of circuits -- only such logical concepts as are either necessary to have a complete system or highly convenient because of the frequency with which they occur and the influence they exert in the relevant mathematical situations. "*

Another early reference comes from 1962 in the description of project STRETCH in which Werner Buchholz states a guiding principle as follows:

> *"The objective of economic efficiency was understood to imply minimizing the cost of answers, not just the cost of hardware. This meant repeated consideration of the costs associated with programming, compilation, debugging, and maintenance, as well as the obvious cost of machine time for production computation. ... A corollary of this principle was the recognition that complex tasks always entail a price in information (and therefore money) and that this price is minimized by selecting the proper form of payment -- sometimes extra hardware, sometimes extra instruction executions, and sometimes harder thought in developing programming systems."*

Ever since the first stored program computer was designed 40 years ago, microarchitecture and compiler technology have both evolved tremendously. Some of these changes are: different ratios of processor speed and memory speed, increasing significance of chip boundaries and improved compiler technology. At each step in this evolution, computer designers have attempted to strike a balance in the placement of functionality. At various times certain functions have been moved from hardware to software and vice versa, and this movement is likely to continue. It has been this change in *technology,* not a change in *design philosophy* which has caused tradeoffs to be optimized in different ways.

In this paper, we have explored the notion of employing advanced software and hardware techniques for exploiting fine grained parallelism in general purpose instruction streams. By providing more background for the analysis of hardware/software tradeoffs, we can better understand what the limits of single instruction stream processors are and better assess how to most cost-effectively allocate resources within a computer system. We have identified three closely related concepts and explored their effectiveness. They are dynamic scheduling, speculative execution and basic block enlargement. We have shown that through the application of these techniques, more parallelism can be exploited than is generally recognized to be present in general purpose instruction streams.

We have presented simulation results showing realistic processors exploiting these three ideas achieving speedups approaching six on non-scientific benchmarks. Given the complexity of the hardware required, some might consider this an upper bound on the performance of single instruction stream processors. Actually, we consider this more of a **lower** bound. There are many unexplored avenues which should be able to push the degree of parallelism even higher. First, we have not fully optimized the use of the techniques analyzed. Better branch prediction could have been used and the implementation of basic block enlargement could have been improved. Also, better compiler technology holds great promise for allowing more parallelism to be exploited. Wider multinodewords put more pressure on both the hardware **and** the compiler to find more parallelism to exploit.

## References

[BeNe62] Bell, C. G., and Newell, A., editors, *Computer Structures: Readings and Examples,* McGraw-Hill, 1971.

[Buch62] Buchholz, W., editor, *Planning A Computer System,* McGraw-Hill, 1962.

[BuGv46] Burks, A. W., Goldstine, H. H., and von Neumann, J., "Preliminary discussion of the logical design of an electronic computing instrument," report to the U.S. Army Ordnance Department, 1946 (reproduced in [Taub63], [BeNe71]).

[Fish81] Fisher, J. A., "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers,* vol. C-30, no. 7, July 1981.

[Hwu87] Hwu, W. W., "HPSm: Exploiting Concurrency to Achieve High Performance in a Single-chip Microarchitecture," *Ph.D. Dissertation,* University of California, Berkeley, December, 1987.

[HwPa87] Hwu, W. W., and Patt, Y. N., "Checkpoint Repair for Out-of-order Execution Machines," *Proceedings, 14th Annual International Symposium on Computer Architecture,* June 2-5, 1987, Pittsburgh, PA.

[HwPa86] Hwu, W. W., and Patt, Y. N., "HPSm, A High Performance Restricted Data Flow Architecture Having Minimal Functionality," *Proceedings, 13th Annual International Symposium on Computer Architecture,* Tokyo, June 1986.

[JoWa89] Jouppi, N. P., and Wall, D. W., "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines," *Proceedings, Third International Conference on*

*Architectural Support for Programming Languages and Operating Systems,* Boston, MA, April 3-6, 1989, pp. 272-282.

[Kell75] Keller, R. M., "Look Ahead Processors," *Computing Surveys,* vol. 7, no. 4, December 1975.

[MeSP88] Melvin, S. W., Shebanow, M. C., and Patt, Y. N., "Hardware Support for Large Atomic Units in Dynamically Scheduled Machines," *Proceedings, 21st Annual Workshop on Microprogramming and Microarchitecture,* San Diego, CA, November 1988.

[PaHS85] Patt, Y. N., Hwu, W. W., and Shebanow, M. C., "HPS, A New Microarchitecture: Rationale and Introduction," *Proceedings, 18th Annual Workshop on Microprogramming,* December 3-6, 1985, Asilomar, CA.

[PMHS85] Patt, Y. N., Melvin, S. W., Hwu, W. W., and Shebanow, M. C., "Critical Issues Regarding HPS, A High Performance Microarchitecture," *Proceedings, 18th Annual Workshop on Microprogramming,* December 3-6, 1985, Asilomar, CA.

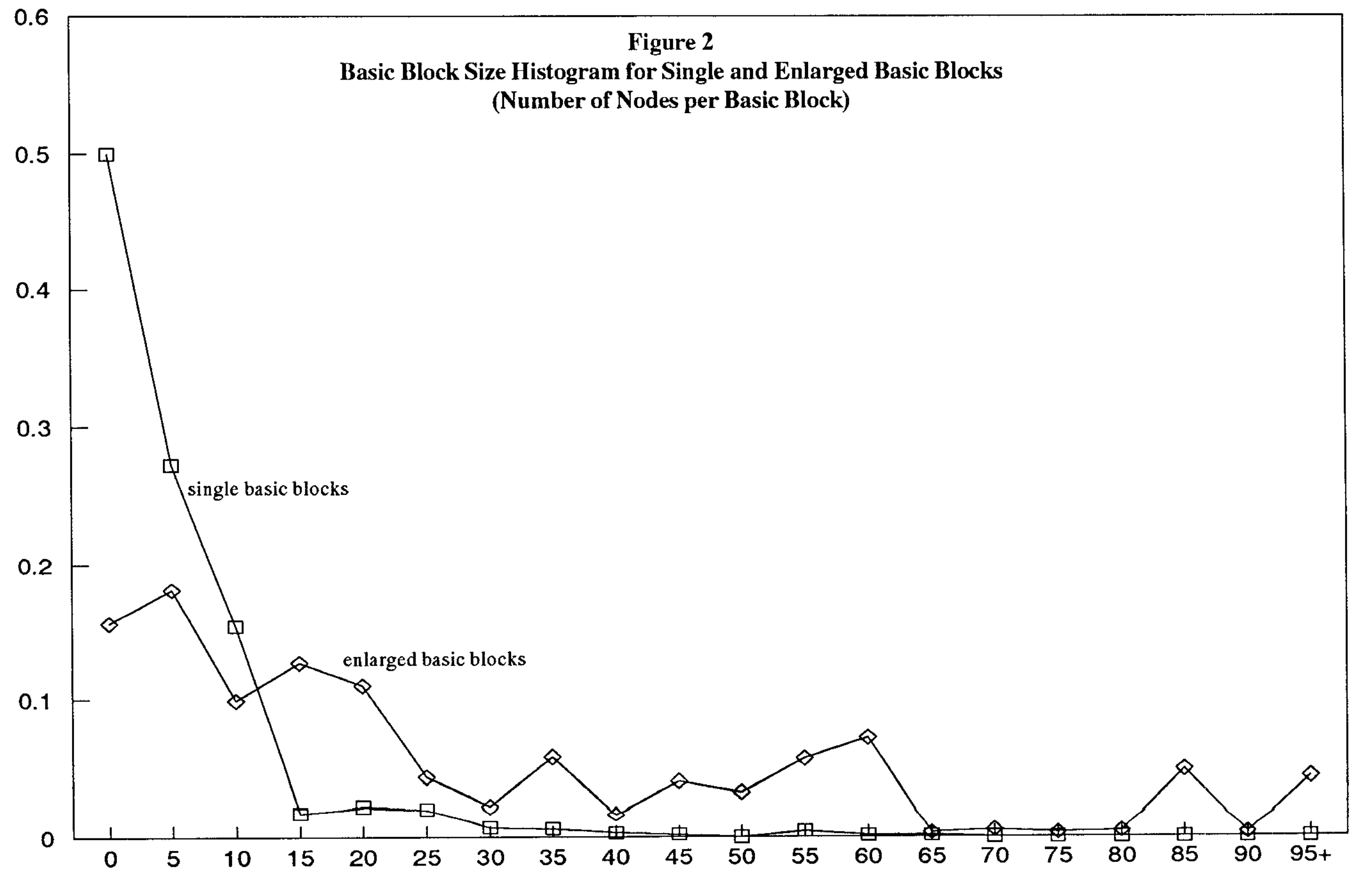
[PSHM86] Patt, Y. N., Shebanow, M. C., Hwu, W., and Melvin, S. W., "A C Compiler for HPS I, A Highly Parallel Execution Engine," *Proceedings, 19th Hawaii International Conference on System Sciences,* Honolulu, HI, January 1986.

[SmJH89] Smith, M. D., Johnson, M., Horowitz, M. A., "Limits on Multiple Instruction Issue," *Proceedings, Third International Conference on Architectural Support for Programming Languages and Operating Systems,* Boston, MA, April 3-6, 1989, pp. 290-302.

[SmLH90] Smith, M. D., Lam, M. S., and Horowitz, M. A., "Boosting Beyond Static Scheduling in a Superscalar Processor," *Proceedings, 17th Annual International Symposium on Computer Architecture,* Seattle, WA, May 28-31, 1990, pp. 344-353.

[Taub63] Taub, A. H., editor, "Collected Works of John von Neumann," vol. 5, pp. 34-79, The Macmillan Company, New York, 1963 (excerpt [BuGv46] reproduced in [BeNe71]).

[Toma67] Tomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development,* Vol. 11, 1967, pp. 25-33.



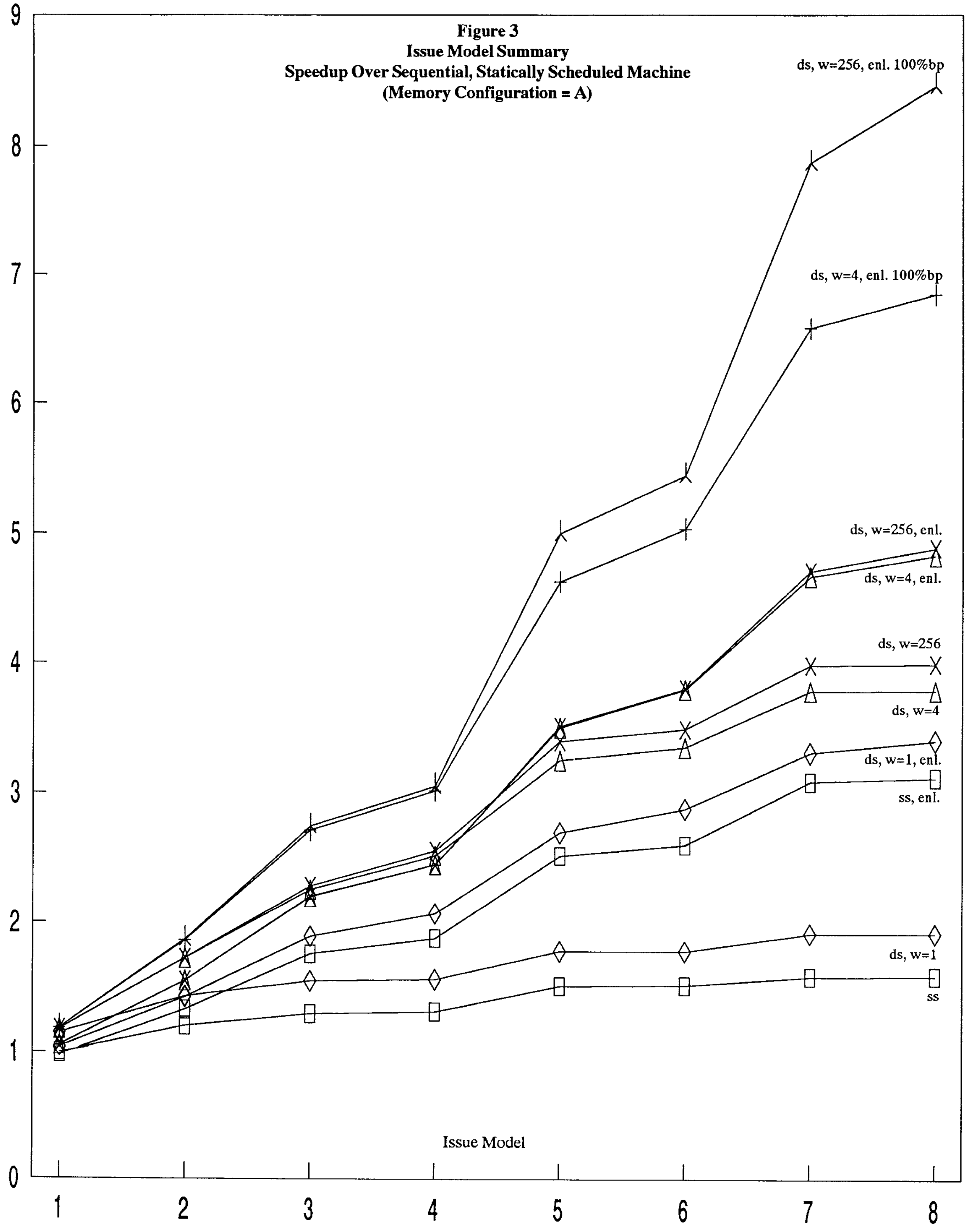


Figure 3
Issue Model Summary
Speedup Over Sequential, Statically Scheduled Machine
(Memory Configuration = A)
ds, w=256, enl. 100%bp
ds, w=4, enl. 100%bp
ds, w=256, enl.
ds, w=4, enl.
ds, w=256
ds, w=4
ds, w=1, enl.
ss, enl.
ds, w=1
ss
Issue Model
0
1
2
3
4
5
6
7
8
9

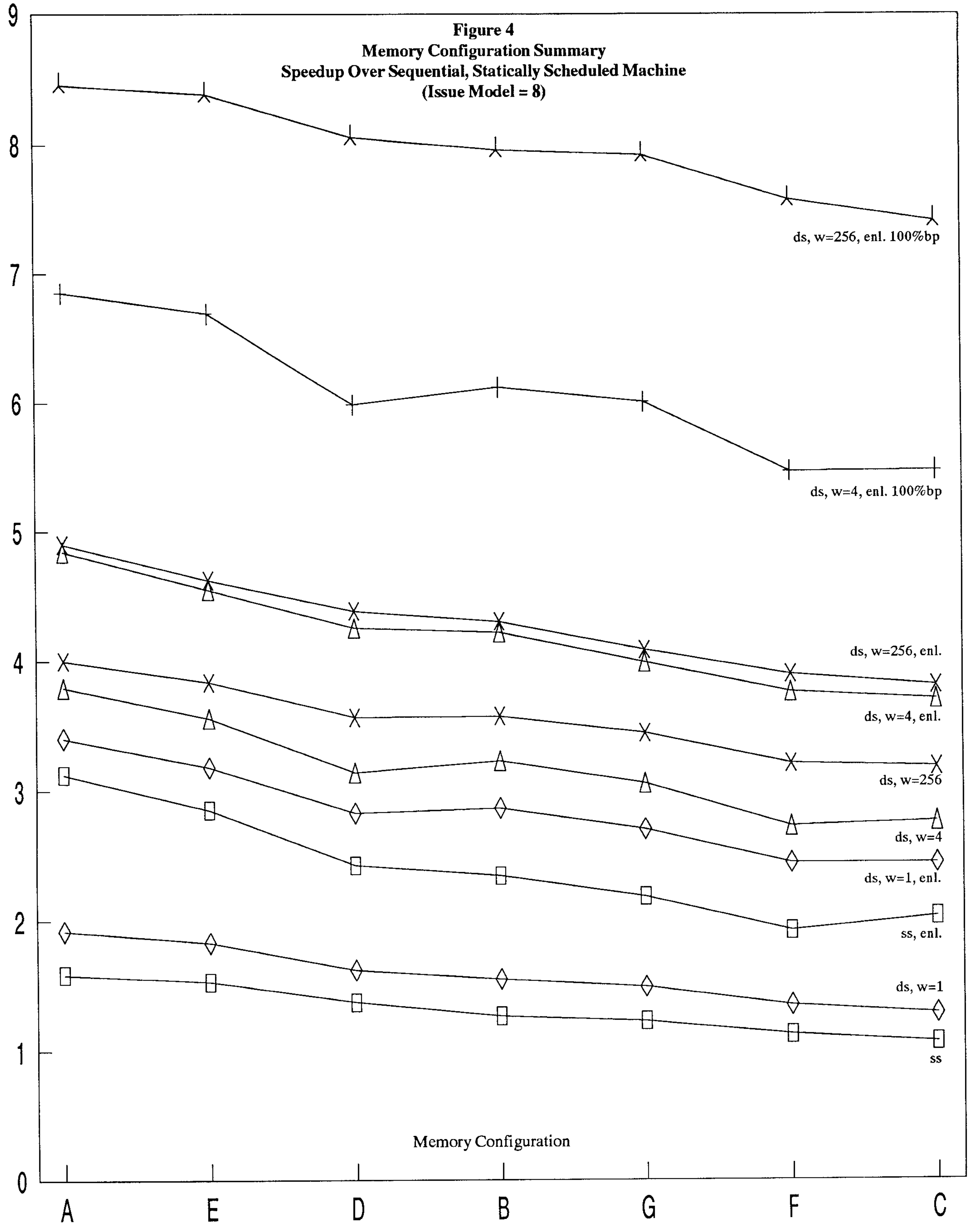


Figure 4
Memory Configuration Summary
Speedup Over Sequential, Statically Scheduled Machine
(Issue Model = 8)
9
8
7
6
5
4
3
2
1
0
ds, w=256, enl. 100%bp
ds, w=4, enl. 100%bp
ds, w=256, enl.
ds, w=4, enl.
ds, w=256
ds, w=4
ds, w=1, enl.
ss, enl.
ds, w=1
ss
Memory Configuration
A
E
D
B
G
F
C

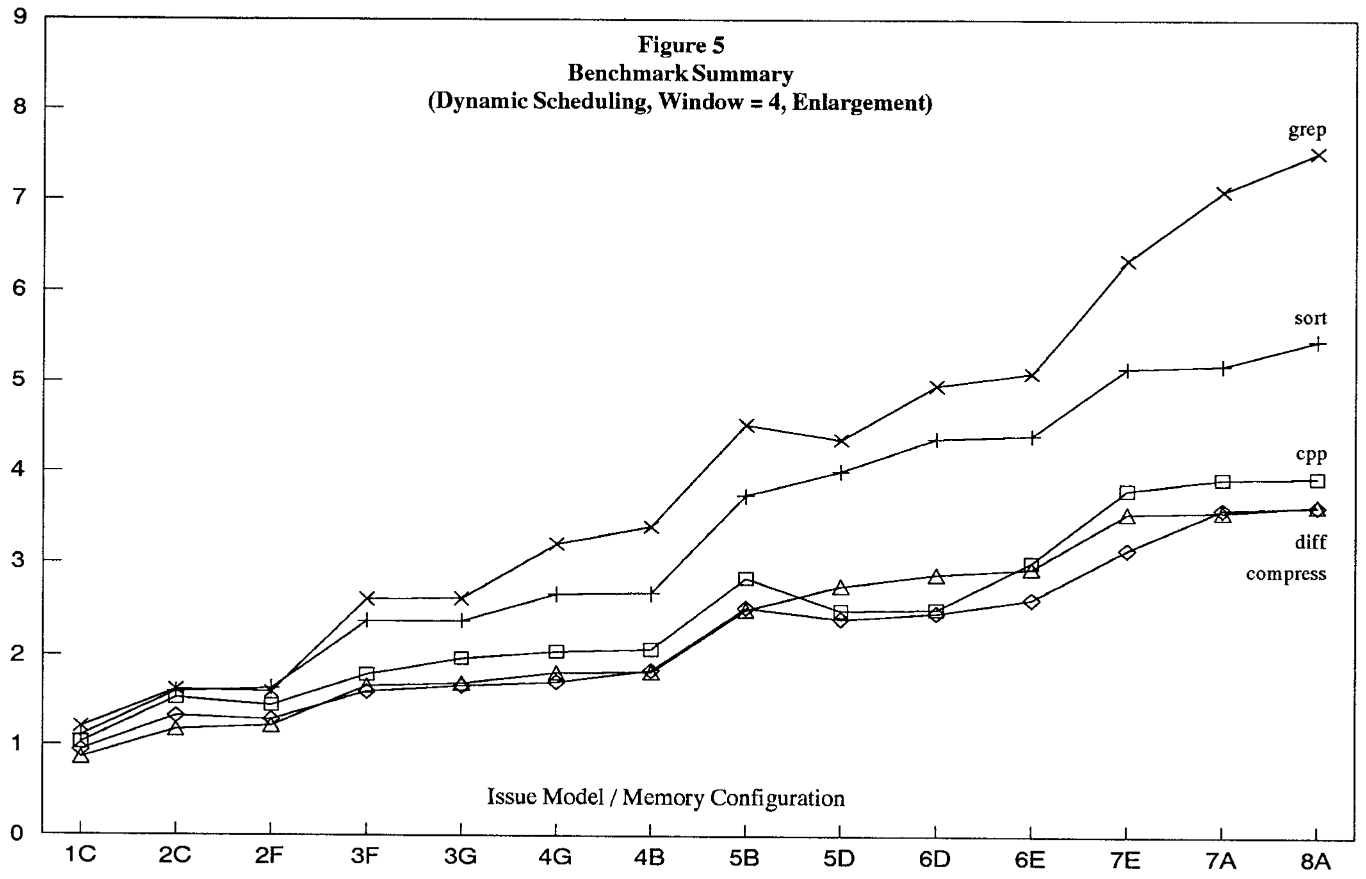


**Figure 5**
**Benchmark Summary**
**(Dynamic Scheduling, Window = 4, Enlargement)**

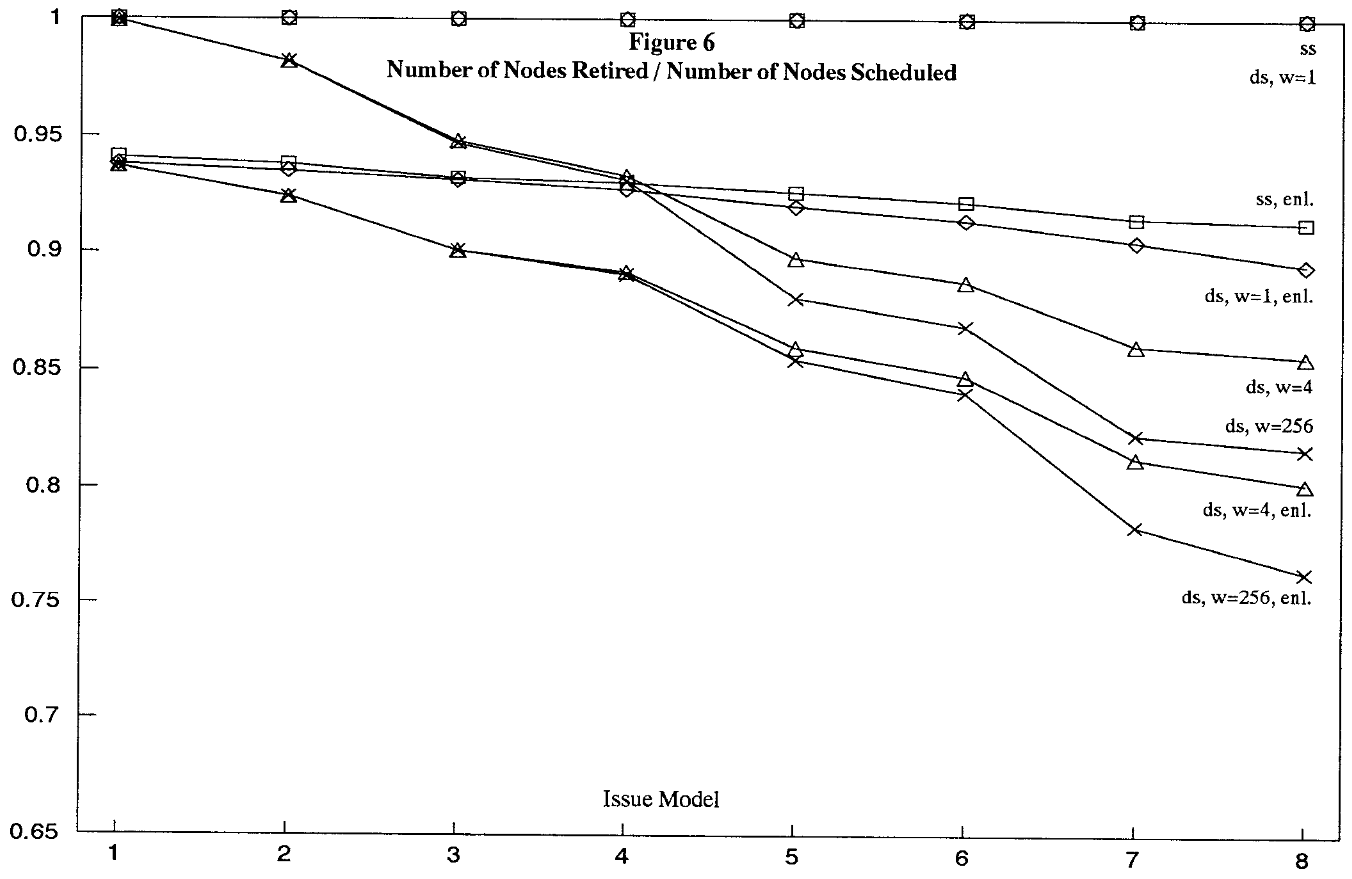


**Figure 6**
**Number of Nodes Retired / Number of Nodes Scheduled**