Measuring the Parallelism Available for Very Long Instruction Word Architectures

ALEXANDRU NICOLAU AND JOSEPH A. FISHER

Abstract — **Long instruction word architectures, such as attached scientific processors and horizontally microcoded CPU's, are a popular means of obtaining code speedup via fine-grained parallelism. The falling cost of hardware holds out the hope of using these architectures for much more parallelism. But this hope has been diminished by experiments measuring how much parallelism is available in the code to start with. These experiments implied that even if we had infinite hardware, long instruction word architectures could not provide a speedup of more than a factor of 2 or 3 on real programs.**

These experiments measured only the parallelism within basic blocks. Given the machines that prompted them, it made no sense to measure anything else. Now it does. A recently developed code compaction technique, called trace scheduling [9], could exploit parallelism in operations even hundreds of blocks apart. Does such parallelism exist?

In this paper we show that it does. We did analogous experiments, but we disregarded basic block boundaries. We found huge amounts of parallelism available. Our measurements were made on standard Fortran programs in common use. The actual programs tested averaged about a factor of 90 parallelism. It ranged from about a factor of 4 to virtually unlimited amounts, restricted only by the size of the data.

An important question is how much of this parallelism can actually be found and used by a real code generator. In the experiments, an oracle is used to resolve dynamic questions at compile time. It tells us which way jumps went and whether indirect references are to the same or different locations. Trace scheduling attempts to get the effect of the oracle at compile time with static index analysis and dynamic estimates of jump probabilities. We argue that most scientific code is so static that the oracle is fairly realistic. A real trace-scheduling code generator [7] might very well be able to find and use much of this parallelism.

Index Terms — **Memory antialiasing, microcode, multiprocessors, parallelism, trace scheduling, VLIW (very long instruction word) architectures.**

I. Introduction

In this paper we describe experiments we have done to empirically measure the maximum parallelism available to *very long instruction word* (VLIW) architectures. The most familiar examples of VLIW architectures are *horizontally microcoded CPU's* and some very popular specialized *scientific processors,* such as the floating point systems AP-120b and FPS-164.

Very long instruction word architectures take advantage of fine-grained parallelism to speed up execution time. However, in contrast to vector machines and traditional multiprocessors, no currently available machines use this architecture for great amounts of parallelism. A user in any practical environment is doing well if he obtains a factor of 2 or 3 speedup over sequential execution. Why are these machines not dramatically more parallel? One probable reason is that the popular wisdom has it that a factor of 2 or 3 is all the fine-grained parallelism that is there to exploit. A chief contributor to this belief was a set of experiments done in the early 1970's. They measured the fine-grained parallelism available under the hypothesis that there was *infinite hardware available* to execute whatever parallelism was found. Unfortunately, all they could find was a factor of 2 or 3.

We believe these experiments, done for a somewhat different domain than VLIW architectures, were far too pessimistic. We will explain why shortly. In this paper we report on experiments we have done which we think more directly address this question for VLIW architectures.

A. Very Long Instruction Word Architectures

The defining properties of VLIW architectures are: 1) there is one central control unit issuing a single wide instruction per cycle; 2) each wide instruction consists of many independent operations; 3) each operation requires a small statically predictable number of cycles to execute. Operations may be pipelined.

Restrictions 1) and 3) distinguish these from typical multiprocessor organizations.[1]

Since it is nearly impossible to tightly couple very many highly complex operations, the underlying sequential architecture of a VLIW will invariably be a *reduced instruction set computer* or *RISC* [16]. Thus, the instruction set will typically consist of register to register operations, with memory references being simple loads/stores without complex addressing modes. This will greatly simplify the scheduling part of the compiler.

VLIW machines might have large numbers of identical functional units. When they do, we do not require that they be connected by some regular and concise scheme such as shuffles or cube connections. A tabular description of the somewhat ad hoc interconnections suffices for our purposes.[2] This makes the use of VLIW machines very different from machines with regular interconnection structures and/or complex hardware data structures.

Manuscript received February 17, 1984; revised July 13, 1984. This work was supported in part by the National Science Foundation under Grants MCS-81-06181 and MCS-81-07646, in part by the Office of Naval Research under Grant N00014-82-K-0184, and in part by the Army Research Office under Grant DAAG29-81-K-0171.

A. Nicolau is with the Department of Computer Science, Cornell University, Ithaca, NY 14853.

J. A. Fisher is with the Department of Computer Science, Yale University, New Haven, CT 06520.

[1] VLIW architectures do not fit neatly into many of the taxonomies of parallel processor organization.

[2] We rely heavily on the compiler to schedule data movements and (when feasible) parallel memory fetches as well as operation execution. Thus, the interconnections between the various processing units need not be very regular, as we expect only the compiler (not humans) to have to deal with them.

A VLIW machine, the ELI (enormously long instructions), is being designed and built at Yale University. The machine has instruction words in excess of 1000 bits, and has the ability to take advantage of considerable fine-grained parallelism. A detailed discussion of VLIW architectures and the ELI machine can be found in [9].

1) Why Have these Machines Not Been Built for Massive Parallelism? We believe that the following statements, valid or not, characterize the feelings most architects have about VLIW architecture machines. We feel this has limited the amount of parallelism that has been built into VLIW architectures.

1) Hand coding machines even as wide as those built today is nearly impossible. Hand coding significantly wider machines is out of the question. (By wider, we mean having more instruction level parallelism, thus more instruction word bits.)

2) The machines are very irregular and inelegant. It is nearly impossible to prove anything about them or algorithms for them. This has probably contributed to a lack of academic interest in them, and has led to a general lack of knowledge about them.

3) Code generation is impossible or very difficult, inhibiting the use of reasonable high level languages.

4) Experiments have indicated a general lack of available parallelism.

We agree fully with reasons 1) and 2). Coding VLIW machines is a horrible task (ask anyone who has tried). And they are not very pretty to look at.

But we disagree with points 3) and 4). We attempt to demonstrate in this paper that the earlier experimental measures should not be taken as upper bounds on available parallelism. And we feel that *trace scheduling* [8], a recently developed global code generation technique, has allowed an effective VLIW machine compiler to be built. Code generation for VLIW machines differs from the ordinary in that it does large scale code rearrangement in order to pack operations efficiently into wide instructions. If this process, called *compaction,* is not done, the code will usually be intolerably slow.

B. What Were the Earlier Experiments?

Tjaden and Flynn [18] and Foster and Riseman [10] did the following.

- They took actual machine language execution streams from normal programs running on various IBM/CDC machines.
- They broke the streams up into basic blocks.
- They asked: Given infinite hardware (they were thinking in terms of an infinite number of CDC/IBM functional units), how much faster would the programs have been executed? We may assume that instruction issue is instantaneous.

When one considers the hardware that inspired these experiments, this is quite a reasonable question to ask. These machines use run time hardware scheduling, assigning operations to functional units, to take advantage of fine-grained parallelism. The hardware really cannot know what to do in the face of a conditional jump. It has to play safe, not being able to do a data flow analysis of the program, and it has to wait until the conditional jump settles before writing variables with possibly incorrect values.

Various machines, including the IBM STRETCH, 360/95, and the Manchester MU-5 [14], attempted some amount of calculation past jumps, but they did not do anything irreversible until the jump settled. This seemed a limited benefit.

But what would one do if one *really* had infinite hardware? One would have been running the same program on two identical machines, and whenever a conditional jump came along, one would let the two of them take different paths. One would then just throw out the wrong one when the test settled. In a companion to one of the above experiments, that is exactly what was measured [17]. Although they found extensive speedups might be available, they dismissed the exploitation of parallelism by bypassing conditional jumps as impractical. They note that obtaining even a tenfold speedup mandated that one would have to start with $o(2^{16})$ machines, a rather unacceptable hardware cost.

C. So What Was Wrong with these Experiments?

There are several problems with the above experiments. To start with, they were all heavily biased towards hardware oriented schemes of identifying parallelism. As such, they spent a great deal of effort in adapting their model to closely emulate the idiosyncrasies of the particular hardware which motivated their experiments, reducing the generality of the results as measures of parallelism available in ordinary programs. Even more important, this approach effectively limited the measured parallelism to basic blocks. Dynamically overcoming basic block boundaries on a large scale was deemed completely unrealistic and they did not foresee any possibility of bypassing large numbers of conditional jumps *statically* at compile time. Given the unavailability of good global compaction methods at the time, their attitude was quite natural.

The domain from which the sample programs were taken further reduces the utility of these results for our purposes. All experiments dealt with seemingly randomly chosen programs from several applications areas. Most of them appear to be heavily data-driven (e.g., compiler, sorter, string matching), some with complex flow of control. Such algorithms are not likely to be amenable to effective compile time global compaction techniques, and thus are of limited interest from our perspective. Furthermore the data-dependent nature of control flow in these programs negatively affects the practicality of the results.

D. Other Previous Experiments

Many suggestions for improved parallelism location have been reported in the literature, often accompanied by empirical measures of effectiveness. Important work done by D. Kuck's group at the University of Illinois [3], [5], [11], has measured the actual parallelism obtainaorable by such algorithm transformations as recurrence solution and tree height reduction. In particular, in [12] it is shown that using the compilation techniques developed as part of the Parafrase project at the University of Illinois, 16 or more processors could be kept busy. However these measurements involve efficiency and practical considerations which are quite different from the *bounds* we are discussing here.

E. Trace Scheduling

People who hand code similar architectures would not consider restricting themselves to basic blocks while doing code rearrangements. It is commonly possible to move an operation up past a jump, making one branch shorter at the expense of the other. Similarly, operations may move down into one or both branches. These operations may be done "for free" in unused fields of the target instruction.

Trace scheduling is a technique for replacing the block-by-block compaction of code with the compaction of long streams of code, possibly thousands of instructions long. The major characteristics of trace scheduling are the following.

1) Dynamic information is used at compile time to select streams with the highest probability of execution.

2) Pre- and postprocessing allows the entire stream to be compacted as if it were one basic block. *This is the key feature of trace scheduling. By using good scheduling heuristics on long streams, large quantities of far reaching parallelism may be efficiently identified.* Thus, we avoid a case by case search for code motions, which would be far too expensive on such large streams and would be unlikely to yield much parallelism.

3) Preprocessing is done to prevent the scheduler from making absolutely illegal between block code motions, i.e., those that would clobber variables which are live off the trace. The constraints on code motion are encoded into the data precedence graph via edges from the conditional jump off the trace to any later operation that could clobber the variable. This makes those constraints look just like the usual data precedence constraints that a basic block scheduler would have to cope with. The scheduler is then permitted to behave just as if it were scheduling a single basic block. It pays no attention whatsoever to block boundaries.

4) After scheduling has been completed on a stream, the scheduler has made many code motions which, while potentially legal, will not correctly preserve jumps from (or rejoins to) the stream to the outside world. To make these code motions actually legal, a postprocessor inserts new code at the stream exits and entrances to recover the correct machine state outside the stream. Without this ability, available parallelism would be unduly constrained by the need to preserve jump boundaries.

5) The most frequently executed code beyond the stream, including the new state recovery code, is then compacted as well, possibly producing more state recovery code.

Eventually, this process works its way out to code with little probability of execution, and more mundane compaction methods are used which do not cause the possible code growth of the state recovery operations.

Trace scheduling will not do well on code which is dominated by highly unpredictable jumps (e.g., parsing loops or tree searches). Experience with scientific code (our target domain) shows, however, that one can make a good guess at the direction of most jumps in this type of code. These guesses may be derived using a tool to run debugged code on samples of data, or may be programmer supplied.

Inner loops of code present a special opportunity for code compaction. *Software pipelining* is a technique used by hand coders to increase parallelism. It involves rewriting a loop so that pieces of several consecutive original iterations are done simultaneously in each new iteration. Although interesting work has been done towards automating software pipelining [11], [13], the techniques do not seem to lend themselves well to more complex types of loops. Trace scheduling, on the other hand, may be trivially extended to do software pipelining on any loop. It is sufficient to simply unroll the loop for many iterations. This will yield a stream, which may be compacted as above. All the intermediate loop tests will now be conditional jumps in the stream; they require no special handling beyond that always given conditional jumps. While that may be somewhat less space efficient than is theoretically necessary, it can handle arbitrary flow of control within each old loop iteration, a major advantage in attempting to compile real code. The extra space is not of major importance here. A detailed description of trace scheduling can be found in [8], while an overall discussion of its actual implementation as part of the Bulldog compiler (a compiler for VLIW machines developed at Yale University) is given in [7].

F. Using an Oracle to Measure Available Parallelism

Given trace scheduling, it is clear that the restriction to basic blocks is not realistic. How may we obtain a more reasonable empirical upper bound on what we could do with very cheap hardware? A reasonable attack is to assume that an *oracle* is present to guide us at every conditional jump, telling us which way the jump will go each reference, and resolving all ambiguous memory references.

Loosely speaking, the parallelism found by the oracle is equivalent to the execution of an idealized data flow machine (assuming infinite resources, a language which resolves ambiguities, and no resources/synchronization induced constraints on parallelism). In fact, trace scheduling attempts to achieve *at compile time* an effect similar to that of data-flow processors [2], [6].

1) True Data Precedence and the Oracle: Another limitation of the earlier experiments of Tjaden and Riseman is that they used register assignments to determine data precedence. This is unrealistic since these assignments are done by a compiler to sequential streams, and impose artificial data-precedence relations.[3] Data precedence should be based upon *actual uses of values*. To handle this, the *single identity principle,* used in many parallel processing applications, should be followed. It mandates that even though a programmer may have reused a variable name (programmers do that for clarity and for storage allocation), we give the new use of the variable a new name if the two definitions never reach any common use.

Arrays present very special data-precedence considerations. A topic of general interest in parallel processing is the disambiguation of indirect memory references [4].[4] For example, in the loop

[3]Register allocation techniques for trace scheduling compilers is a subject of current research.

[4]This is a problem only when scheduling at compile time. At runtime no ambiguous references can exist.

```
for I := 2 to N do
    A(I) := A(I - 1) + tot;
    A(I + 2) := A(I + 1) + tot;
    tot := tot + B(I)
    end
```

we would need to know that $A(I)$, $A(I-1)$, $A(I+1)$, and $A(I+2)$ were different from each other to have proper code motion freedom. A hand coder would not think twice about permuting the first two lines. More sophisticated memory disambiguation involves an analysis of the possible range of a variable's values, and the solution of diophantine equations. The situation is made more complex in the presence of conditional jumps. Such a disambiguator has been implemented at Yale as part of the Bulldog compiler.

Since we are finding an empirical upper bound on available parallelism, we also ask our oracle to resolve all indirect references. That means that code which chases down pointers, say, would have a lot of parallelism exposed that we would never be able to find in any practical environment. But, once again, the static nature of the scientific code we are considering makes this not such an unrealistic assumption. In any event, it is an upper bound on available parallelism that we are seeking here.

While we ask our oracle to overcome many of the limitations to parallelism, the obstacle that we do not eliminate is that of data precedence. We are forced to wait until values have been calculated before we use such values. Naturally, we could again ask our oracle for the value of all computed variables, but that would be much less realistic than the tasks we have asked the oracle to do so far. That would also trivialize our results since we could just ask for all output variables at the outset, and do all our computation in one step!

II. The Design of the Experiment

In the process of designing our experiment several decisions had to be made.

- We needed to choose a *language level* at which the parallelism would be found.
- *Sample programs* within which to find parallelism had to be chosen.
- We had to devise an *oracle* that would eliminate the inhibition of parallelism due to conditional jumps and ambiguous memory references.

A. Experimental Language Level

For the purpose of our experiment we have dealt with programs represented in a language we call N-ADDRESS CODE or NADDR. NADDR is being used as the intermediate language for several code compaction projects at Yale University, and may be compiled from several high level languages. Similar to the 3-address codes used as compiler intermediate languages [1], NADDR is at the right level for this type of experimentation. A higher level language would not have presented enough granularity for our purposes. A specific assembly language would have burdened the experiment with unnecessary detail and might have biased our results by losing some of the parallelism in the binding to machine constructs.

1) NADDR Code: An instruction in NADDR is written as an operator, followed by one or more operands. The meaning and number of the operands are determined by the operator. There are no expressions in NADDR, so in compiling from high level languages, temporaries are introduced to buffer intermediate operands. For example, the Fortran statement $A = B*(C + D)$ would appear as shown in Fig. 1. Fig. 2 shows (a) a sample Fortran program and (b) its equivalent n-address version. NADDR contains only one-dimensional arrays, multidimensional arrays being linearized by our translator. Some of the directives in the figure are obscure, and meant for the interpreter's eyes only.

In order to simplify the bookkeeping required by our measurements, we have assumed that each instruction takes exactly one time unit to execute. This assumption avoids the need to deal with specific complex architectures; it's removal would have a minimal effect on our results.

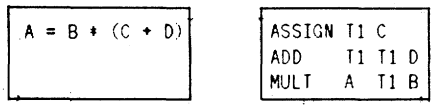
B. The Sample Programs and How They Were Chosen

In our experiment we have focused on numerical analysis applications, although a few other programs have been included for comparison purposes. Numerical applications were selected because they are a logical application of the compaction techniques that suggested this experiment.

- Numerical analysis programs are very CPU intensive and could greatly benefit from any achievable speedup. In real time applications, the speedup may allow the execution of certain programs which would otherwise not be feasible.
- Since good compaction methods are expensive to run, they are most useful in cases where the programs compacted have relatively large life spans. Many numerical analysis programs fall into these categories.
- Numerical analysis programs as a group lend themselves well to global compaction techniques. Conditional transfers of control and pointer chasing are the main inhibitors of parallelism. In this respect, numerical applications do well since the control structures of these programs tend to be relatively simple and the conditional jumps are heavily biased towards one of the possible paths. Most array references are found within loops in which the index is a simple increment or something equally tractable. These may be successfully analyzed for potential parallelism at compile time. In short, numerical code is typically not very data driven.

Another advantage comes with the non-data-driven property of most scientific code. The available parallelism we measured is not greatly affected by the data we happened to choose. Most of these programs will execute nearly the same streams of instructions no matter what the data. Thus, we did not have to fear biasing our results by the use of atypical data.

Table I contains a list of the programs we used. Most of these were chosen as being typical numerical analysis code in everyday use. The only modification made to the original programs was the replacement of I/O statements with reads from and writes to array buffers. This is a standard method for dealing with I/O in parallel applications, removing the need to deal with the intrinsic sequentiality of these operations. In some applications these factors might dominate the running time, but those are not the applications we are interested in.



(a) (b)

Fig. 1. Example of NADDR code representation. (a) Fortran statement. (b) NADDR code representation.

```
(DCL L7 LONG 1 (3.0))
(DCL CON1 LONG 1 (1))
(LASSIGN L7 CON1)
(DCL S-8 INT 1 (0))
(DCL CON5 INT 1 (5))
(IASSIGN S-8 CON5)
(LABEL L5)
(ISUB S-8 S-8 CON1)
(DCL CON0 INT 1 (0))
(DCL T1 INT 1 (0))
(ILT T1 S-8 CON0)
(TRUEGO T1 L6)
(LABEL L1)
(DCL T2 INT 1 (0))
(INT-LONG T2 L7)
(DCL CON-1 INT 1 (-1))
(DCL T3 INT 1 (0))
(IADD T3 T2 CON-1)
(DCL L8 REAL 5 (0 0 0 0 0))
(DCL T4 DOUBLE 1 (0))
(DOUBLE-LONG T4 L7)
(DASSIGN (OFFSET L8 T3) T4)
(LADD L7 L7 CON1)
(GOTO L5)
(LABEL L6)
(DCL L9 REAL 1 (0))
(DCL CON0D0 REAL 1 (0.0))
(DASSIGN L9 CON0D0)
(LASSIGN L7 CON1)
(IASSIGN S-8 CON5)
(LABEL L11)
(ISUB S-8 S-8 CON1)
(DCL T5 INT 1 (0))
(ILT T5 S-8 CON0)
(TRUEGO T5 L12)
(LABEL L10)
(DCL T6 INT 1 (0))
(INT-LONG T6 L7)
(DCL T7 INT 1 (0))
(IADD T7 T6 CON-1)
(DCL T8 DOUBLE 1 (0))
(DADD T8 (OFFSET L8 T7) L9)
(DASSIGN L9 T8)
(LADD L7 L7 CON1)
(GOTO L11)
(LABEL L12)
(LABEL L1)
```

```
      Dimension A(5)
      DO 10 I = 1,5
10      A[I] = I

      SUM = 0

      DO 20 I = 1,5
        SUM = SUM/5
      STOP
      END
```

(a) (b)

Fig. 2. Sample program. (a) Input Fortran program. (b) NADDR output of the translator.

TABLE I
SAMPLE PROGRAM DESCRIPTIONS

Program Name	Description
RFALS1	Calculates roots of scalar functions using the Regula Falasi method
RFALS2	Calculates roots of scalar functions using the Regula Falasi method, with check
EXPNT	Sample numerical analysis computation. Computes a negative exponent using two methods.
GRAM	Stabilized Gram-Schmidt orthogonalization.
ODE4	Solves the differential equation Y' = Y on the interval [0 , 10], using an implicit two step method.
EXTRAP	Evaluates the derivative of a given function at a given point using extrapolation.
BISECT	Computes a root of a polynomial using the bisection method.
NEWT1	Computes the root of a polynomial using Newton's method.
NEWT2	Computes the root of a polynomial using Newton's method. (similar to NEWT1, but higher precision).
JACOBI	Solves a system of linear equations using the Jacobi iteration method.
FIXPT	Calculates a fixed point approximation for a given function.
ROFERR1	Sample numerical analysis computation. (Exemplifies the influence of round-off errors).
EULER	Approximate solution to the initial value problem: Y'=Y, Y_0=1 on the unit interval using Euler's method.
GAUS	Solves a linear equations system using the Gauss-Seidel iteration Method.
CHEBY	Finds a maximum point for a polynomial with evenly spread roots, on the interval [-1 , 1].
SINFFT	Computes a one dimension sine transform using a one dimensional complex fast Fourier transform.
SPOFA1	Factors a real symmetric positive definite matrix (Choleski). Input size(20x20)
SPOFA2	Same as above but for a larger input matrix (50x50).
RS	Input size (10x10)
SQRDC	QR factorization. Input size(40x20)
SSVDC	Singular value decomposition of a real matrix. Input size(8x8)
SGEFA	Factors a real matrix by Gaussian elimination. Input size(4x4)

C. Optimal Scheduling Using the Oracle

1) What the Oracle Had to Do: In this experiment, operations are scheduled in cycles at compile time in such a way that true data precedence is the only constraint to parallelism. The conceptual model we used for resolving dynamic questions at compile time involves an oracle. The oracle allows us to always know, while scheduling, which way a jump would go. It allows us to completely unwind the code, and it tells us what locations all indirect references are to, so that we may do code motions without being unsafe.

In order to simulate such an oracle we used two major modules: an interpreter and a scheduler. The interpreter accepts the NADDR generated from high-level code and executes it, instruction by instruction. The stream produced is passed to the scheduler which places each instruction at the earliest possible level for execution, based on the dependencies between the current instruction and the previously scheduled ones. The "two pass" nature of this process gives us our oracle. When an instruction is ready to be scheduled, the code has executed all the way up through that instruction. Thus, all branches have been determined, and all earlier

indirect memory references have been resolved. The ability to "know" which way jumps go with 100 percent accuracy enables the oracle to create an optimal schedule for the run. This is a result of only scheduling the operations that are actually executed (no operations from the unused branches are ever scheduled by the oracle).

Both the interpreter and the scheduler were compiled MACLISP running on a DEC-20 under TOPS-20.

2) Coping with Very Large Instruction Streams: It was still necessary to follow data-dependency constraints. To do this, many scheduling schemes use a DAG (directed acyclic graph) representation of the program, in which the vertices represent instructions and the edges express the dependency relation between them. However, we were experimenting with rather large programs, and were scheduling entire execution traces, not just basic blocks. Building a full DAG to represent the dependency relations would have required far too much space.

Fortunately, such a representation was not necessary for our purposes. In our model, in which the only dependencies are data dependencies, it suffices to build a symbol table to keep track of the schedule level at which a variable was last read/written. Then when an instruction is considered for placement in the schedule, the variables read and written by the instruction are checked against the stored information. The instruction is scheduled at the first level consistent with all the data dependencies of the instruction. The infinite resources hypothesis guarantees that the instruction may be legally scheduled there, and data precedence assures that it may not be scheduled any earlier. Thus, the bound we obtain is optimal in the sense of this experiment. The space overhead required to store last read/last written information in this scheme is minimal since we keep in effect only a "front line" of the DAG, which is dynamically updated in the process of building the schedule. The details of the scheduling algorithm which achieves this are given in Fig. 3, together with the rules for legal references of data-dependent variables. Another advantage of this approach is that instructions can be scheduled on the fly, without the need to keep the whole schedule in memory. The only information required to establish the level of an instruction is provided by the last read/written levels of the variables used by the instruction. Besides not keeping the entire DAG in memory, there was no reason to keep the schedule itself. We simply waited until the end, and noted the largest cycle number at which any variable had been written. That was the length of the schedule.

III. Results

A. What was Measured

For the programs in our database, each running a single set of data, we measured the following.

- *Speedup ratio* — The ratio between the number of cycles required for the sequential execution of the program, and the number of cycles required for the parallel execution of the compacted version of the program. This is the measurement we were most interested in.
- *Average speedup per basic block* — Defined exactly as above, except with compaction limited to inside basic blocks. This was measured for comparison to earlier experiments done by others, who measured only this. We also wanted to see the difference between global and local available parallelism in our domain.
- *Number of added variables* — The number of variables introduced in the process of renaming. Renaming may be used to allow some calculations to proceed beyond a conditional jump, and be conveniently thrown away when the jump goes the "wrong way." This enhances parallelism by removing some artificial dependencies. Our code did this even when it was clear that it would not provide added parallelism, so this is an upper bound measurement.
- *Maximal level width* — The maximal number of computational resources (e.g., adders, multipliers, etc.) used by the most resource intensive cycle of the compacted schedule. Since no attempt was made to even out the use of resources, this is very much an upper bound measurement.

The last two measurements were added for interest; we did not attempt to draw any conclusions from them.

B. Potential Global Speedup Ratio

Looking at Table II, we see that *the available global speedups found by our experiment ranged from below 4 to 988 times increase in execution speed.*

Many of the programs were limited only by the size of their data[5] (e.g., programs 17 and 18 in Table II). If anything, the data we usually picked were smaller than those encountered in realistic applications since we had to simulate the code.

Several of these programs (1, 2, 3, 5, 11, 12, 13 in Table II) are rather sequential in nature. Thus, one would expect less potential parallelism, and that is the case. This is probably parallelism that a compacting compiler could take advantage of, though, since it does not generally involve disambiguating memory references.

The programs that exhibit the largest parallelism (4, 6, 17, 18, 22 in Table II) operate on arrays. Intuitively, array processing presents greater opportunities for parallelism, and these results bear this out. The magnitude of the speedups clearly indicates that they do not result from the parallelization of trivial parts of the programs (such as the initialization of arrays), but rather that the parallelism is intrinsic to the whole computation process. Here the disambiguation of memory references is particularly important, so it is more problematical whether practical compaction methods will be able to approach the upper bound. However, the amount of available parallelism is large enough to encourage the development of sophisticated (and costly) disambiguation techniques.

C. Basic Block Compaction

We found that when compaction was limited to basic blocks only, the available speedup was almost always less than a factor of 2.5. This is consistent with earlier experiments [10], [18]. The difference between this measure and the global speedup is dramatic enough by itself to account for a large measure of the pessimism that has been felt about VLIW architectures. If trace scheduling, or some other global compaction technique, can recover a large share

[5]While the program control flow was not influenced greatly by the data contents, size made a difference in the amount of parallelism available.

(a) Rules for the Preservation of Data Dependency:

1. An operation reading variable V_i, can be scheduled for execution no earlier than the next cycle after the one in which V_i was last written. That is, if V_i was last written at cycle L_j, it could be read at cycle L_j+1 at the earliest.
2. An operation writing variable V_i, can be scheduled for execution no earlier than the cycle in which V_i was last read. That is, if V_i was last read at cycle L_j, it could be written at cycle L_j at the earliest. Note that if renaming is used, this rule becomes superfluous.

(b) Inner Loop of Scheduling Algorithm:

```
Read(operation)
WHILE operation # EOF DO BEGIN

   FOR V_i IN Read-List(operation) DO BEGIN

       L_i  = Cycle-Last-Written(V_i) + 1, /* By rule 1 above */
       Cycle-List  = Cycle-List || L_i,      /* Add L_i to possible
                                                cycle list */
       END,


   FOR V_i IN Write-List(operation) DO BEGIN

       L_i  = Cycle-Last-read(V_i),        /* By rule 2 above */
       Cycle-List  = Cycle-List || L_i,      /* Add L_i to possible
                                                cycle list */
       END,


   Max-Cycle  = MAX(Cycle-List),                /* Find earliest possible
                                                   Cycle */

   Insert-in-Schedule(operation,Max-Cycle),     /* Schedule operation at
                                                   earliest safe cycle */

 /* Reset last read/written information for all variables in operation */
   FOR V_i IN Read-List(operation) DO
       Update-Last-Read(V_i, Max-Cycle),

   FOR V_i IN Write-List(operation) DO
       Update-Last-Written(V_i, Max-Cycle),

   Read(operation),
   End;
```

Fig. 3. Instruction scheduling. (a) Data dependency rules. (b) Algorithm.

TABLE II
RESULTS

	1 RFALS1	2 RFALS2	3 EXPNT	4 GRAM	5 ODE4	6 EXTRAP	7 BISECT	8 NEWT1	9 NEWT2	10 JACOBI	11 FIXPT	12 RFALS1	13 RFALS2	14 EXPNT	15 GRAM	16 ODE4	17 EXTRAP	18 BISECT	19 NEWT1	20 NEWT2	21 JACOBI	22 FIXPT
STREAM LENGTH	2047	841	9042	40530	1983	1859	1477	4605	696	480	201	1375	31111	460	50000	3822	3219	17634	23225	19784	32877	1333
MIN SCHED LENGTH	190	205	3015	41	324	28	70	103	18	23	76	52	5394	23	1436	22	28	58	1557	501	805	26
MAX LEVEL WIDTH	258	30	4	3234	238	198	80	1503	288	58	6	123	20	56	4293	292	479	2669	1518	3203	3118	192
TEMPS ADDED	1934	735	7026	33955	1630	1501	1376	4373	633	348	162	1148	23953	348	39986	3127	2419	13519	18963	16089	25631	862
NUMBER OF BLOCKS	62	77	2009	5238	310	275	62	202	32	26	32	212	7138	26	9998	5.18	498	2718	2874	2328	5014	268
AVG BLOCK LENGTH (BEFORE)	33.02	10.92	4.5	7.74	6.40	6.76	23.82	22.80	21.75	18.46	6.28	6.49	4.36	17.69	5.00	7.38	6.46	6.48	8.08	8.49	6.55	4.97
AVG BLOCK LENGTH (AFTER)	13.35	6.69	3.0	3.95	3.60	3.56	11.35	6.44	6.13	3.31	3.38	4.73	2.86	7.96	3.06	3.71	2.03	2.00	3.61	3.28	3.44	2.64
SPEEDUP PER BLOCK	2.47	1.63	1.5	1.96	1.78	1.90	2.10	3.54	3.55	5.58	1.86	1.37	1.53	2.22	1.64	1.99	3.18	3.23	2.24	2.58	1.90	1.88
POTENTIAL GLOBAL SPEEDUP	10.77	4.10	3.0	988.5	6.12	66.39	21.1	44.72	38.67	20.87	2.65	26.44	5.77	20	34.82	173.73	114.96	304.03	14.91	39.49	40.84	51.26

of this difference, then there is real hope for using VLIW architectures for significant parallelism.

IV. Can this Parallelism be Exploited?

In these experiments we have endeavored to find an upper bound on the available parallelism, which may be obtained by simple program rearrangement. As such these results are very encouraging with respect to the potential for fine-grained parallelism exploitation on VLIW architectures. Still, since we were finding a bound, we assumed an oracle and a hardware model which were unrealistic in several respects. We will now discuss the extent to which the bound can be approached by a trace scheduling compiler for a real-life VLIW machine.

A. Hardware Limitations

Although it does not directly involve the oracle, the most unrealistic aspect of the model is the assumption of unlimited hardware resources. Clearly, the parallelism ultimately used will be only as large as permitted by the width of the real-life machine.

Even though the resources required by any given program are obviously not infinite, and are even reasonable, they tend to vary widely and are often dependent on the size of the data. Thus, we are not likely, for efficiency reasons, to build machines which will accommodate the largest possible requirements we may envision. As a result, we are bound to run into programs whose potential parallelism far exceeds that of the available hardware. In these cases, however, VLIW machines can be expected to offer extremely graceful degradation of performance because of the fine granularity of the parallelism used by these machines and because of the ability of trace scheduling to naturally deal with resource conflicts (by encoding them as data dependencies). This is a major advantage of our approach over other methods of parallelism exploitation (e.g., vector machines).

B. How Unrealistic is the Oracle?

Riseman and Foster demonstrated that a straightforward hardware mechanism which obtained even a small portion of the available parallelism would engender an unacceptable hardware cost. The only viable alternative is to try to statically exploit parallelism by compile time global scheduling. The oracle we have postulated did just that, but used two unrealistic assumptions.

- We can *always* predict which way conditional jumps will branch.
- We can *always* predict whether two memory references collide.

While a compiler cannot possibly hope to do this, the real question is: Can it do well enough to exploit a sizable fraction of the exposed parallelism?

1) Trace Scheduling as a Conditional Jump Predictor: Trace scheduling by its very nature attempts to approximate the oracle's prediction of conditional jump directions. The underlying assumption for the success of this process is that almost all jumps branch in a predictable direction almost all of the time. As we mentioned previously, programs in our target domain (scientific computation) have simple and mostly data-independent flow of control, with conditional jumps often biased towards one branch. Parallelism enhancement transformations (e.g., loop unwinding) will only tend to increase this effect. Because of this, we can expect a trace scheduling compiler to come close to the oracle in its branch predictions.

For other types of code (e.g., Systems) containing highly unpredictable flow of control, trace scheduling could not hope to approximate the oracle's predictions. Because of this, we did not include any such programs in our samples since the potential parallelism, however large, would have been meaningless for our approach.

2) Memory Disambiguation as a Reference Conflict Predictor: Our compiler uses conventional and unconventional flow analysis techniques in an attempt to prespecify as accurately as possible at compile time which indirect references access the same address location during the execution of the program. Of course, disambiguation cannot hope to achieve good results (i.e., come close to the oracle's predictions) on references that are very data dependent. For example, pointer chasing down a linked list or dereferencing involving input parameters are often intractable using static disambiguation.[6] The ineffective memory antialiasing of such references will result in spurious data dependencies, and inhibit parallelism.

Fortunately, though, indirect memory references in scientific code are almost always array references, with relatively simple indexes.[7] As a result, the antialiasing system is usually able to disambiguate them effectively, making the oracle predictions quite realistic. As in the case of trace scheduling, parallelism enhancing techniques will not interfere with memory disambiguation. In fact, the potential for successful disambiguation may increase. For example, in the case of loop unwinding, indirect references are generated which differ only by a constant and which are therefore trivial to disambiguate.

C. Substantiating our Claims

As reported upon in [7], J. Ellis, J. Ruttenberg, and the authors of this paper have implemented a trace scheduling compiler at Yale University. Based on our experience with it, we informally divide a test suite of source programs[8] into three broad classes.

Type P1 — Those which (we believe) stand a chance of obtaining significant parallel speedups using an ordinary multiprocessor or a vector machine.[9] Examples of these include an FFT, convolution, and matrix multiply.

Type P2 — Those which appear on the surface to have potential parallelism but are too irregular or interwoven to fit into Type P1. These include a prime number sieve and LU-decomposition.[10]

[6]Sometimes indirect references can be seen not to conflict even when unpredictable parameters are involved in the index calculations.

[7]This property of scientific code has been recognized and used by many other researchers. See, for example, the work of D. Kuck and U. Banerjee.

[8]Somewhat different from the set of programs reported upon here.

[9]Doing so might still require significant programmer effort.

[10]A small algorithmic transformation makes LU-decomposition suitable for a vector processor. We use the straightforward statement of the algorithm. It evidently took years before people in the scientific programming community noticed this transformation, even though this code was the subject of much attention.

Type S — Those which seem too sequential to allow any significant speedup due to parallelism. Examples of these include various series summations, dot products, iterative solvers, etc. This is a place where both vector machines and the Bulldog compiler may be helped by the work of D. Kuck's group at the University of Illinois [15]. Their source transformer can probably solve the recurrence and improve performance on these types of programs.

We believe that Types P2 and S are much more common in real life than Type P1. Since Type P1 is also found in the inner loops of programs of Types P2 and S, it might seem sufficient (e.g., for signal processing) to only handle Type P1. But we believe that even among most applications with Type P1 code in the inner loops, not enough of the running time is in the inner loops to justify special hardware. For example, if 80 percent of the code fits very well, and even if parallelism reduces that nearly to zero, the maximum possible speedup is under a factor of 5.

The general results we obtained on the test bed of programs we currently run through the Bulldog compiler are as follows.

Type P1 — We find all the parallelism the hardware will allow, *without the programmer having to alter the natural expression of the algorithms,* as is customary for with other approaches.

Type P2 — We again find all the parallelism the hardware will allow. We find it even if the inner loops contain conditional flow of control and data-precedence between loop iterations. We find it even with widely scattered references to array elements.[11]

Type S — We generally get a factor of 5 or more speedup. These are programs that we believe would be impossible to speedup at all with other processors.

V. Conclusions

We have shown that a large degree of functional unit parallelism exists in scientific code. Some code has dramatic speedup potential (approaching a factor of 1000) available. A sizable fraction of this parallelism can be exploited by a trace scheduling compiler in conjunction with a smart antialiasing mechanism.

Raw hardware cost has now become the inexpensive component in computing systems. This reality has spawned much research into the problem of formulating algorithms to use parallel hardware effectively. VLIW architectures present a form of parallelism not usually considered in parallel processing studies. This is true partly because of the pessimistic results of early basic block experiments. We hope that our experiments constitute a strong rebuttal of that pessimism. They clearly demonstrate that fine-grained VLIW type parallelism deserves much more attention than it has received.

[11]Our individual control over memory accesses, and our ability to differentiate the banks being referenced, allow us to maintain a high memory bandwidth in situations that would reduce other processors to uniprocessor speed. This is perhaps the most encouraging fact about VLIW's and the Bulldog compiler.

References

[1] A. V. Aho and J. D. Ullman, *Principles of Compiler Design.* Reading, MA: Addison-Wesley, 1977.

[2] Arvind and V. Kathail, "A multiple processor data flow machine that supports generalized procedures," in *Proc. 8th Annu. Symp. Comput. Arch. ACM (SIGARCH),* May 1981, vol. 9, no. 3, pp. 291–302.

[3] U. Banerjee, S. C. Chen, D. J. Kuck, and R. A. Towle, "Time and parallel processor bounds for Fortran-like loops," *IEEE Trans. Comput.,* vol. C-28, pp. 660–670, Sept. 1979.

[4] U. Banerjee, "Speedup of ordinary programs," Dep. Comput. Sci., Univ. Illinois, Urbana, IL, Tech. Rep. UIUCDS-R-79-989, Oct. 1979.

[5] S. C. Chen and D. J. Kuck, "Time and parallel processor bounds for linear recurrence systems," *IEEE Trans. Comput.,* vol. C-24, pp. 701–717, July 1975.

[6] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *Proc. 2nd Annu. ACM–IEEE Symp. Comput. Arch.,* 1974, pp. 126–132.

[7] J. A. Fisher, J. R. Ellis, J. Ruttenberg, and A. Nicolau, "Parallel processing: A smart compiler and a dumb machine," to be published.

[8] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.,* vol. C-30, pp. 478–490, July 1981.

[9] ——, "Very long instruction word architectures and the ELI-512," Dep. Comput. Sci., Yale Univ., New Haven, CT, Tech. Rep. 253, 1982; see also, *Proc. 10th Annu. Int. Arch. Conf.,* Stockholm, June 1983.

[10] C. C. Foster and E. M. Riseman, "Percolation of code to enhance parallel dispatching and execution," *IEEE Trans. Comput.,* vol. C-21, pp. 1411–1415, Dec. 1972.

[11] D. J. Kuck, Y. Muraoka, and S. -C. Chen, "On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup," *IEEE Trans. Comput.,* vol. C-21, pp. 1293–1310, Dec. 1972.

[12] ——, "Measurements of parallelism in ordinary FORTRAN programs," *Computer,* vol. 7, pp. 37–46, 1974.

[13] L. Lamport, "The parallel execution of DO loops," *Commun. ACM,* vol. 17 pp. 83–94, Feb. 1974.

[14] J. K. F. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *Computer,* vol. 17, pp. 6–22, Jan. 1984.

[15] D. A. Padua, D. J. Kuck, and D. H. Lawrie, "High speed multiprocessors and compilation techniques," *IEEE Trans. Comput.,* vol. C-29, pp. 763–776, Sept. 1980.

[16] D. A. Paterson and C. H. Sequin, "A VLSI RISC," *Computer,* vol. 15, pp. 8–21, Sept. 1982.

[17] E. M. Riseman and C. C. Foster, "The inhibition of potential parallelism by conditional jumps," *IEEE Trans. Comput.,* vol. C-21, pp. 1405–1411, Dec. 1972.

[18] G. S. Tjaden and M. J. Flynn, "Detection and parallel execution of independent instructions," *IEEE Trans. Comput.,* vol. C-19, pp. 889–895, Oct. 1970.

Alexandru Nicolau received the B.A. degree in computer science from Brandeis University, Waltham, MA, in 1980 and the M.S. and Ph.D. degrees from Yale University, New Haven, CT, in 1981 and 1984, respectively.

He is currently an Assistant Professor in the Department of Computer Science, Cornell University, Ithaca, NY. His research interests include parallel computation, computer systems architecture, and optimizing compilers.

Joseph A. Fisher received the A.B. degree in mathematics from New York University, New York, NY, in 1968, and the M.S. and Ph.D. degrees in computer science from the Courant Institute of Mathematical Sciences, New York University, in 1976 and 1979, respectively.

He has been at Yale University, New Haven, CT, since 1979, where he is an Associate Professor. His research interests include very long instruction word architectures, parallel processing, automated design of computers, and microprogramming.

Dr. Fisher was the Program Chairman of the ACM–IEEE Fifteenth Annual Workshop on Microprogramming in 1982, and a Board Member of the IEEE Computer Society from 1981 to 1982. He recently received the Presidential Young Investigator Award.