PARALLEL PROCESSING
M. Cosnard, M.H. Barton and M. Vanneschi (Editors)
Elsevier Science Publishers B.V. (North-Holland)
© IFIP, 1988

SOME DESIGN IDEAS FOR A VLIW ARCHITECTURE FOR SEQUENTIAL-NATURED SOFTWARE

Kemal Ebcioğlu
IBM, Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, New York 10598

Abstract

In this paper, we describe a Very Long Instruction Word architecture, now being designed at the IBM T.J. Watson Research Center, which is intended to achieve good performance not only in scientific code, but also in sequential, non-numerical code. Communication delays between processing elements are minimized via a single shared register file with a large number of ports. To perform well on programs with unpredictable branches, the architecture features decision-tree shaped instructions that allow multiway branching, and that allow operations to be executed conditionally depending on where the instruction branches to. To add to the parallelism achievable via existing compilation techniques for VLIW architectures, we have developed a compilation technique called *pipeline scheduling*, which is an extension of the "doacross" and "dopipe" techniques proposed for multiprocessors by D. Kuck's group. This technique can initiate a new iteration of an inner loop (possibly containing arbitrary if-then-else statements and conditional exits) on every clock period whenever dependences and resources permit.

Background

It appears that a certain amount of the inherent parallelism in ordinary programs is fine-grain; in fact, fine-grain parallelism often appears to be the only type of parallelism available in a large body of non-scientific programs. Unfortunately, the major trends in parallel architecture are not geared toward exploiting such irregular fine-grain parallelism, except for the data flow paradigm [Dennis 74,80, Arvind and Ianucci 83], which in turn is not too useful for existing software because it requires algorithms to be rewritten in a special functional language, and incurs some overhead in inherently sequential code [Gajski et al. 82]. SIMD machines like the GF-11, [Beetem, Denneau and Weingarten 85], systolic arrays like the Warp machine [Arnould et al. 85], and the more general-purpose vector supercomputers like the Cray [Russell 78] tend to work well on an important but limited class of scientific problems, but fail to achieve speedup on problems that do not belong to their domain; for example, a typical scientific supercomputer reduces to a uniprocessor on program segments where vectorization is not possible. The problem of transforming ordinary programs to run on MIMD multiprocessor architectures has also received much attention [Kuck 78, Allen and Kennedy 84], and while a speedup by a factor of thousands appears to be possible for certain scientific problems with a large degree of inherent parallelism [Veidenbaum 85], the speedup results with non-numerical algorithms have not been very promising: for example, [Lee, Kruskal,and Kuck 85] report a typical speedup of 1.3-1.5 on simple nonnumerical algorithms such as binary search and merging, using their Parafrase compiler for an MIMD architecture (assuming that the algorithms are not rewritten in a parallel way). It should be noted that program restructuring techniques as in the University of Illinois Parafrase compiler, are best capable of exploiting the *coarse grain* parallelism in scientific code, which does not seem to be common in non-numerical code. While there is a modest amount of fine-grain parallelism in non-numerical programs, practical MIMD architectures cannot exploit it well, because even a few cycles of communication overhead can cause the speedup to be much less than ideal.

The VLIW architecture

Following the advances in microcode compaction techniques in the recent years after the introduction of the trace scheduling technique [Fisher 79],[1] an architecture called the Very Long Instruction Word (VLIW) architecture has been proposed by J. Fisher [Fisher 82], which is specifically intended for exploiting the modest fine-grain parallelism inherent in ordinary high level language programs. VLIW machines, as the name implies, have an unusually long instruction word, on the order of 500-1000 bits or more, and can perform many simultaneous operations in a single instruction. VLIW architectures are roughly based on the idea of compiling high level languages directly into horizontal microcode. Although a VLIW architecture has a single control mechanism as in a uniprocessor, it is distinct from a uniprocessor, because it can execute the equivalent of many uniprocessor instructions in a single cycle, thus surpassing the so-called *Flynn limit* for uniprocessors [Flynn

[1] Trace and trace scheduling are trademarks of Multiflow Computer, Inc..

66].[2] It is also different from the typical SIMD architecture, because it can simultaneously execute different operations in each of its processing elements.[3]

State of the art in VLIW compilation techniques

We will now briefly go over the status of the compilation and optimization techniques that were developed for horizontal microarchitectures and VLIW architectures. Many problems that arise in program optimization, such as minimizing execution time, or minimizing program size, are known to be computation intensive, or to be unsolvable, depending on how the problem is posed [Machtey and Young 78, Rogers 67]. The early approaches to microprogram optimization nevertheless did attack the problem of minimizing program size and/or execution time [Agerwala 76]. The early algorithms were confined to the optimization of basic blocks, and those that did achieve optimality had to rely on time-consuming enumerative methods, for example, [Yau, Schowe and Tsuchiya 74] describe a branch and bound technique for finding an optimal schedule for a given straight line microcode segment. Moreover, the studies on real programs, for example [Foster and Riseman 72], showed that a parallelism of only about 1.7 could be achieved, if the optimizations were limited to individual basic blocks. Although enumerative optimization methods such as this branch and bound technique may have become practical now, considering the smallness of basic blocks and the raw computing power now available, the early approaches to microcode compaction nevertheless did not have much promise of achieving tangible parallelism, because of the limited nature of the parallelism within basic blocks.

For obtaining higher parallelism than that which was available within basic blocks, J. Fisher [Fisher 79] took the approach of aggressively overlapping the operations from different basic blocks, for example, executing operations from the most probable clause of a forthcoming if-then-else statement concurrently with the execution of the operations of the current basic block. Fisher's technique, called *trace scheduling*, first chooses, with the aid of some heuristic, a particular execution path or *trace* within an acyclic flow-graph. It then compacts the operations in the trace into microinstructions, as if the whole trace were a single basic block (taking care not to prematurely schedule assignments to variables that are live off the trace, so that the program does not give wrong results even if the trace is not followed during execution after all), and finally makes the necessary adjustments in the other parts of the program affected by the trace compaction, for example it provides copies of portions of the original code for those paths that were joining and leaving the original trace. Once the first trace is done with, another trace which is disjoint from the original trace is picked and compacted, etc., until no uncompacted code remains. Nested loops are handled by applying trace scheduling on the inner loop body and then treating the inner loop as a single node within the enclosing loop, which can be done in reducible flow graphs. The implement the trace compaction, a list scheduling algorithm can be used [Adam, Chandy, and Dickson 74] which has a worst case running time which is only quadratic in the number of micro-operations in the trace. This scheduling technique does not guarantee optimality, but gives good results in practice. A problem that limits the practical applicability of trace scheduling for non-numerical code, is that the probabilities of the conditional branches must be specified by the programmer, or must be determined by actually running an uncompacted version of the program; moreover, conditional branches must have a high probability of branching in one direction rather than the other. When program execution does not follow the trace picked first by the scheduler (say path A), and follows a different path B, then the compacted program may perform poorly, compared to how it would perform if path B were picked first by the scheduler (this is true even for a machine with unlimited resources, if the operations outside a trace are not allowed to move into the trace after the trace has been compacted into machine code). When combined with the additional technique of unrolling inner loops prior to trace scheduling, the trace scheduling technique was reported to achieve about an order of magnitude of parallelism in scientific code [Fisher 82, Fisher and O'Donnell 84]. When not aided by loop unrolling, and when we lift the assumption that only scientific code will be executed, we have found in our (very preliminary) experiments that the trace scheduling family of compaction techniques can achieve a practically implementable parallelism of about 3, which we still feel is a good base speedup to improve upon.

Fisher's approach was somewhat complex, especially when correction were applied to the paths joining and leaving the original trace. A. Nicolau [Nicolau 85] greatly simplified the ideas behind the trace scheduling

[2] It might be argued that the multiple operation elements available in a VLIW architecture are similar to multiple functional units such as in the CDC 6600 [Thornton 64] or IBM 360/91 floating point unit [Tomasulo 67]. But the instruction issue rate available in VLIW machines tends to be greater than multiple functional unit, pipelined architectures. For example, if we disregard the chaining capability of the Cray-1 for vector operations, we see that the CDC 6600 and Cray-1 [Russell 78] architectures can still issue a maximum of one instruction per cycle, even though they have many functional units which can operate in parallel. (But a CDC/Cray-inspired architecture that can simultaneously issue *two* instructions in a single cycle has recently been developed [Goodman et al. 85].)

[3] Assuming an ideal VLIW machine which has an unlimited amount of ALU resources and memory ports, and which can perform complex conditional branches to arbitrarily many targets at each cycle, VLIW machines can theoretically have the same power as a *synchronous* MIMD architecture: an ideal synchronous MIMD configuration containing k identical RISC processors each capable of executing an instruction every cycle, and running k different programs with $n_1,\ldots,n_k$ instructions each, can be translated to a VLIW configuration with a program that contains at most $n_1 \times \ldots \times n_k$ instructions. I.e., each instruction of the VLIW machine represents the entire state of all the processors in the MIMD computer.

·ause it can simultaneously execute different

ımization techniques that were developed for ›lems that arise in program optimization, such ›own to be computation intensive, or to be Young 78, Rogers 67]. The early approaches m of minimizing program size and/or exe- › the optimization of basic blocks, and those tive methods, for example, [Yau, Schowe and an optimal schedule for a given straight line ample [Foster and Riseman 72], showed that tions were limited to individual basic blocks. ınd bound technique may have become prac- omputing power now available, the early ap- ch promise of achieving tangible parallelism. ›.

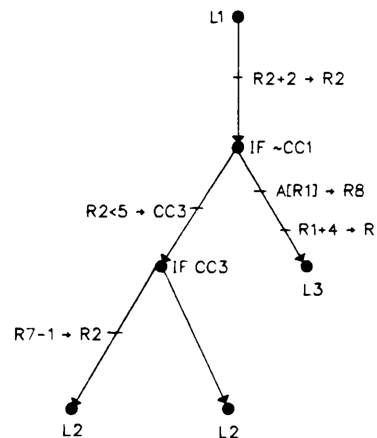
in basic blocks, J. Fisher [Fisher 79] took the ıt basic blocks, for example, executing oper- ›e statement concurrently with the execution alled *trace scheduling*, first chooses, with the an acyclic flow-graph. It then compacts the e were a single basic block (taking care not the trace, so that the program does not give fter all), and finally makes the necessary ad- ompaction, for example it provides copies of eaving the original trace. Once the first trace e is picked and compacted, etc., until no un- trace scheduling on the inner loop body and › loop, which can be done in reducible flow algorithm can be used [Adam, Chandy, and adratic in the number of micro-operations in . but gives good results in practice. A problem umerical code, is that the probabilities of the ıst be determined by actually running an un- ›s must have a high probability of branching does not follow the trace picked first by the e compacted program may perform poorly, the scheduler (this is true even for a machine allowed to move into the trace after the trace additional technique of unrolling inner loops rted to achieve about an order of magnitude 84]. When not aided by loop unrolling, and ted, we have found in our (very preliminary) ques can achieve a practically implementable › improve upon.

ction were applied to the paths joining and ›lified the ideas behind the trace scheduling

architecture are similar to multiple functional units such ›ulo 67]. But the instruction issue rate available in VLIW ctures. For example, if we disregard the chaining capa- ay-1 [Russell 78] architectures can still issue a maximum ›ich can operate in parallel. (But a CDC/Cray-inspired ıs recently been developed [Goodman et al. 85].) resources and memory ports, and which can perform W machines can theoretically have the same power as a containing *k* identical RISC processors each capable of instructions each, can be translated to a VLIW i.e., each instruction of the VLIW machine represents

technique and placed them on a more formal basis, by defining a set of primitive *core transformations* on the programs of a microprogramming language for an abstract VLIW machine with infinite resources. He called this technique *percolation scheduling*. By applying the core transformations, one can repetitively move ("percolate") operations from one microinstruction to a preceding one, and achieve the effect of overlapping operations from different basic blocks, and in particular of trace scheduling. However, percolation scheduling is not limited to compacting operations on a single trace, and so may result in higher performance regardless of whether a particular trace is followed during execution or not. The main advantage of percolation scheduling vs. trace scheduling is the following: Given a machine with sufficient resources and proper architectural support; percolation scheduling, together with renaming optimizations, can be used to parallelize programs with unpredictable conditional branches, by executing operations on *all* forthcoming basic blocks past conditional jumps, as soon as their operands are ready; so a program can perform at a rate close to the optimal data flow speed, regardless of which branch is taken. On the other hand, trace scheduling will tend to perform poorly if the trace that is picked first by the scheduler is not taken during the actual execution of the program.

We will now discuss some core transformations of percolation scheduling as applied to the VLIW machine we will be discussing in this paper. But first, let us briefly explain the form of the instructions of our machine, and their semantics. The instructions of our machine have the form of a directed binary tree as shown below (A formal description of this computation mechanism was given in [Ebcioğlu 87]; in this paper, we will strictly stress the clarity of the exposition rather than formalism).




```
L1:
 ((ADD R2 2 R2)
 (IF   (NOT CC1)
       ((LT R2 5 CC3)
        (IF   CC3  ((SUB R7 1 R2) (GOTO L2))
              ELSE ((GOTO L2))))
       ELSE ((LOAD A R1 R8) (ADD R1 4 R1) (GOTO L3))))
```

The root node of the tree is marked with the label of the instruction. On each node other than the root and the terminal nodes, there is a mark indicating a test on a condition code register (the machine has multiple condition code registers). The tip nodes of the tree are marked with the labels of other instructions in the program, that this instruction can branch to. On each directed edge in the tree, there may be zero or more three-register arithmetic or comparison operations, and memory loads and stores. An instruction is executed in a single clock cycle, as follows: First, the current (i.e. old) values of the condition code registers are examined and a unique path through the tree is selected, starting from the root node and ending at a tip node, as follows (a path from the root to a tip node is called an *i-branch* of the instruction): when a node with a test on a condition code register is reached (called a *test node*), the path continues with the edge that goes to the left, if the test is true for the current value of the condition code register; otherwise, if the test is false, the path continues with the edge that goes to the right, and so on, until a tip node is encountered. After the path is selected, only the operations and memory loads and memory stores on the selected path are executed (in parallel), using the old values of the registers as operands or addresses, and the processor branches to the instruction whose label is indicated at the tip node of the selected path through the tree. The operations and memory loads/stores that

are not on the selected path are not performed. When there is more than one three-register operation or load operation that sets the same destination register on the selected path, the result of the operation that is closer to the tip node takes precedence and actually goes to the destination register at the end of the cycle. A Lisp-list notation for the same tree is given after the example instruction tree. We will later discuss how this mechanism is efficiently implemented in hardware with a short cycle, and answer the questions that may to come the reader's mind about operations that take longer than a single cycle, etc..

In figure 1, we give an informal list of the percolation scheduling core transformations, re-formulated by us to fit the conditional execution mechanism of our machine. (Nicolau's original abstract microprogramming language for percolation scheduling [Nicolau 85] has essentially the same if-then-else branching mechanism, but does not have any conditional execution mechanism; all operations are unconditionally executed in Nicolau's language, i.e., they are all located in the stub edge of the tree. The ability of our machine to execute operations conditionally depending on where the instruction is branching to, is a very important architectural feature from the performance viewpoint; since it allows an instruction to gain headway from the target stream before actually branching to the target. This technique reduces critical paths and loop iteration issue delays in software pipelining, described later).

```
INITIAL PROGRAM:
(1):
 ((IF  cc4   ((GOTO (2))) ELSE ((GOTO (4)))))
(2):
 ((IF  (NOT cc5)  ((ADD x 1 x) (GOTO (6)))
       ELSE       ((ADD x y z) (GOTO (7)))))

MOVE-OP TRANSFORMATION:
(1):
 ((IF  cc4   ((ADD-U x y z) (GOTO (2'))) ELSE ((GOTO (4)))))
(2'):
 ((IF  (NOT cc5)  ((ADD x 1 x) (GOTO (6)))
       ELSE       ((GOTO (7)))))
; old (2) is retained if it still has predecessors

MOVE-CJ TRANSFORMATION:
(1):
 ((IF  cc4   ((ADD-U x y z)
              (IF (NOT cc5) ((GOTO (2''))) ELSE ((GOTO (2''')))))
       ELSE ((GOTO (4)))))
(2''):
 ((ADD x 1 x) (GOTO (6))
(2'''):
 ((GOTO (7))
; old (2') would be retained if it still had other predecessors

DELETE TRANSFORMATION:
(1):
 ((IF  cc4   ((ADD-U x y z)
              (IF (NOT cc5) ((GOTO (2''))) ELSE ((GOTO (7)))))
       ELSE ((GOTO (4)))))
(2''):
 ((ADD x 1 x) (GOTO (6))
;(2'''): (deleted)

UNIFICATION TRANSFORMATION:
(6):
 ((IF  cc2   ((ADD y 1 y) (GOTO (8)))
       ELSE ((ADD y 1 y) (GOTO (9)))))
 becomes
(6):
 ((ADD y 1 y)
  (IF  cc2   ((GOTO (8)))
       ELSE ((GOTO (9)))))
```

Figure 1: Percolation scheduling core transformations (modified by us)

The move-op transformation moves a simple operation, such as (ADD x y z) (meaning $x+y \rightarrow z$) in the example, from one instruction (2), to the edge leading to a tip node of a predecessor instruction (1). If the predecessor

i-branch of the previous instruction (1) writes into x or y, or if the old value of z is live at the beginning of instruction (2) on a path not passing through this particular (ADD x y z) in (2), or if z is used on a path passing through this (ADD x y z) in (2) by some operation in (2) other than (ADD x y z), or if there is an assignment to z between this (ADD x y z) and the root of (2), then the move of this operation (ADD x y z) cannot be made; otherwise the move can be made, to obtain the result shown in the figure.[4] Note that in the process of moving a conditionally executed operation such as (ADD x y z) to a preceding instruction, where it will be executed unconditionally, the operation has to be made uninterruptible (indicated by -U here), in order to prevent an overflow exception that would not have occurred in the original version of the program.[5] The move-cj transformation moves a test node such as "IF (NOT cc5) ...", from an instruction (2') to a preceding one (1), by creating two modified copies of (2'): one, labeled (2''), that acts as if the test (NOT cc5) were true; and another, labeled (2'''), that acts as if the test (NOT cc5) were false. After the move-cj, the predecessor i-branch in (1), instead of branching to (2'), branches to (2'') if (NOT cc5) is true, and to (2''') if (NOT cc5) is false. If the cc5 register used by the test node is set in the predecessor i-branch of the previous instruction (1), the move of the test node, of course, cannot be done. In both the move-op and move-cj transformations, if the instruction which originally contained the operation or test node had another predecessor i-branch, the original copy of the instruction has to be retained in order to preserve program semantics. Then, the operations in the original copy can also be percolated upward through the other predecessors. The delete transformation simply deletes an empty instruction, that has become empty as a result of moving its contents upward. It is the delete transformation that actually reduces the path lengths and causes speedup. There is another transformation called the unification transformation, which acts on a single instruction, and which serves to unite several copies of the same operation after it and its copies have been pushed up along the different branches of, e.g., an if-then-else statement. The unification transformation can be done if an operation is present on both of a twin pair of edges emanating from the same node n in the instruction tree (or if the operation is present on only one of a twin pair of edges, but an imaginary copy of the operation can be inserted in the edge that does not have it, without harming program semantics). The operation and its (possibly imaginary) copy are united and are moved up as a single operation to an edge higher in the tree, namely, the edge coming in to the node n. On-the-fly incremental applications of classical optimizations, such as dead code elimination, copy propagation, or common expression elimination, can be combined with percolation scheduling. Assuming that the target architecture supports conditional operations and if-then-else trees, and that sufficient machine resources are available, a greedy application of percolation scheduling that takes each operation or conditional jump in the order it appears in the sequential version of a loop free program, and moves up each operation or conditional jump as high up as it will go on all paths (attempting unifications before each move-op), will already yield a good schedule, and is sufficient to demonstrate the concept.[6] But scheduling with finite resources, variable length operations, etc., and the related correctness, optimality and complexity issues, are less easy, and we are continuing to do research on these topics.

A further technique for achieving parallelism in compiled code for VLIW machines is unrolling inner loops a few times, and applying the scheduling techniques to the unrolled loop body [Fisher 82]. There have also been other improvements to trace scheduling that have been published in the microprogramming workshops since Fisher's thesis, such as *tree compaction*, [Lah and Atkins 83], which prevents the proliferation of the new, copied nodes generated by the trace scheduling process. But we feel that the impending availability of denser main memory chips, and the use of code explosion control techniques, may at this stage alleviate the code size problem of the trace scheduling family of compaction techniques.

---

[4] Actually, even if the old value of z is live on a path through (2), the move can still be made by changing this definition of z to (ADD x y z'). If this definition (ADD x y z) of z covers all of its uses, i.e., each of its uses use only this particular definition of z, then all of its uses can be renamed as z'; otherwise, a (COPY z' z) operation, that copies z' back to z, can be placed in the old location of (ADD x y z) in (2) to preserve program semantics. Also, if the old value of z at the beginning of instruction (2) covers all of its uses, then the move can again be made by adding a transfer (COPY z z') alongside (ADD x y z) in instruction (1), that copies the old value of z to z', and renaming all uses of the old value of z to z'.

[5] If an overflow-causing addition actually occurs during (ADD-U x y z), z will be set to the special bit pattern ⊥ (bottom), but no exception will take place. Then, if the path through the old location of (ADD x y z) in (2) is actually taken by the program, and z (or the result of any uninterruptible operation that depends on z) is later used as an operand to an interruptible operation, an exception will occur; so some arithmetic error trapping is provided by our architecture. The alternative solution of deferring the exception until the ⊥ value is stored into memory, is not a good one, since the exception could be detected too late in case many variables are allocated in registers. We will describe the hardware support for arithmetic traps later in the paper.

[6] Notice that the code produced by such a greedy application of percolation scheduling, does not execute *all* the operations in the original code, regardless of whether they are useful or not (doing so would be unacceptably inefficient, e.g. in a decision tree type code structure). As soon as it is known that a path is not going to be taken, the code produced by greedy percolation scheduling stops executing operations on that path, so resources are wasted precisely on the operations whose operands become ready, before or at the same time as the condition code(s) that determine the non-necessity of these operations become ready. If the peak parallelism of the problem is so low (as in system or commercial code) that several machine resources are already practically infinite for the problem, wasting resources in this controlled fashion will probably not hurt.

State of the art in VLIW architectures

Now let us review the state of the art in VLIW architectures. A VLIW architecture would best be implemented from scratch, since none of the existing micro-architectures appear to offer the degree of parallelism and flexibility demanded by the task of compiling high level languages. Traditional wide-word microarchitectures like the internal microarchitectures of traditional mainframes, or even general purpose user microprogrammable computers like the Nanodata QM-1 [Nanodata 79], or the Kyoto University QA-1 and QA-2 [Hagiwara et al. 80, Tomita et al. 86], or the Floating Point Systems FPS-164 [Charlesworth 81] are in our opinion not too useful as VLIW's because these machines seem to have been designed more for hand-coding than for specifically compiling high-level languages. Perhaps with the exception of the QA-1 and QA-2, such machine tend to have irregular and non-uniform resources which do not lend themselves well to compiling high level languages. Moreover, none of these machines have the capability of implementing multiway branching in the form of if-then-else trees, which is the natural mechanism for executing several ordinary conditional branch instructions in a single cycle. Nevertheless, until about a year ago, VLIW architectures intended for compiling high level languages were still in the planning stages. But presently, a company called Multiflow already has a machine; and there seems to have been a sudden increase in interest in designing VLIW architectures. J. Fisher, when he was back at Yale, has considered a VLIW architecture called ELI-512, with 8 fixed point and 8 floating point "clusters" each of which has separate banks of registers, ALUs, multipliers, etc.; and a 512 bit instruction word [Fisher 82]. The ELI architecture has essentially a crossbar interconnection within a cluster, but timid interconnections between clusters, and extra cycles are required to move data between clusters. The multiway branching capability of the ELI is of the form: if $test_1$ then goto $target_1$, else if $test_2$ then goto $target_2$, ... else goto $target_n$, which is good for compaction of conditional jump operations on a single trace through the code (but conditional jumps outside the single trace cannot be executed in parallel with the ones in the trace, with this mechanism.) Fisher is now with Multiflow, whose Trace 28/200 computer [Colwell et al. 87] is an ELI-inspired machine, in the sense that it features I-boards for integer operations and F-boards for floating point operations, each of which have separate register banks and ALUs. The Trace 28/200 has an instruction cache, multiple memory banks, a pipelined data memory access technique where a data memory access completes in several cycles, and pipelined ALUs that complete a floating point operation is several cycles. Memory bank disambiguation is done primarily through software. We feel that this is an effective configuration especially for scientific code. A. Nicolau and Kevin Karplus have considered a VLIW architecture called ROPE [Karplus and Nicolau 85], that capitalizes on highly interleaved, slow, but low-cost dynamic memory as the instruction memory. The ROPE architecture does support a multiway branching capability similar to that of percolation scheduling, but it has a very sequential implementation of this branching mechanism; it issues the prefetches (one per cycle) to the separate target instruction streams of a multiway branch, several cycles ahead of the cycle where the actual multiway branch decision is made for choosing a particular target stream for execution. This machine's throughput would consequently not be high on code whose original sequential version is conditional branch intensive, moreover the ROPE prefetching scheme seems to have considerable engineering complexity compared to the alternative of sending the correct target address to an instruction cache. Another related architecture is the polycyclic architecture designed by B. R. Rau, [Rau and Glaeser 81, Rau, Glaeser and Picard 82], which is not a conventional microarchitecture, in the sense that it does not have any registers. Instead there are FIFO-like buffers between the output of every functional unit and the input of every functional unit. The functional units are pipelined. The FIFO-like buffers are useful for, e.g., allowing an operation of iteration n+1 of a software-pipelined loop to proceed and produce a result, even though the result of the same operation in iteration n has not yet been consumed by all destinations of that operation.[7] B.R. Rau is now with a company called Cydrome, whose Cydra 5 (TM) machine is an architecture inspired from the polycyclic architecture [Cydrome 88]. The Cydra 5 has highly pipelined floating point adder and multiplier units, and a high latency but pipelined data memory system. The branching capabilities (i.e., conditional branch throughput rate) of this machine are not clear, but a different compilation technique is used instead to remove conditional branches from the code: If-then-else statements within loops are eliminated by computing the values calculated at *both* the then and else parts and selecting the correct value via a machine operation similar in semantics to the C expression (test?operand1:operand2). This technique converts the loop body to an IFless basic block. Stores to memory in the untaken path of an if-then-else are conditionally disabled. Good software pipelined performance is possible especially in vectorizable inner loops. But due to the basic block model for the inner loop, a single iteration initiation interval (i.e., delay between the initiations of two consecutive iterations) is chosen for a given inner loop during software pipelining (even if the original loop allows different iteration initiation intervals depending on the different paths that may be taken through the loop body). CHoPP [Burke 87] is another scientific supercomputer, which consists of several VLIW processors which share a common memory, where each

[7] It is interesting to contrast Rau's technique with the static data flow paradigm [Dennis 80]. The data flow paradigm can also achieve software pipelining by executing operations from iteration n+1 before finishing all the operations of iteration n, but in static data flow (e.g., implemented with the "acknowledge" method) an operation of iteration n+1 waits until all the result tokens of the same operation in iteration *n* are consumed. Tagged token data flow architectures do not run into this pipeline stalling problem (at the cost of some overhead).

VLIW processor features a multiport register file with full connectivity to 4 integer ALUs, and another, separate, multiport register file with full connectivity to 4 floating point ALUs, apparently two-way conditional branching, instruction cache, and a single port to the shared memory. So there is certainly a rapidly growing interest in VLIW architectures, but the targeted software almost always seems to be scientific code, rather than inherently sequential code, which is the subject of our architecture research.

The pipeline scheduling technique

We will now describe a new VLIW compilation technique that we believe could augment the repertoire of compilation techniques for VLIW architectures.

It is an obvious fact that the speed at which inner loops are executed have a critical effect on the runtime of an algorithm. When each of the inner loop iterations can be independently executed, then vector instructions can often be used in a supercomputer, or the iterations can be allocated to different processors on a MIMD architecture. For loops where some iteration depends on some previous iteration, which appears to be a common case in real programs, executing the iterations in pipelined fashion, e.g., starting a new iteration every cycle, is an attractive way to achieve speedup. In the context of microprogrammable architectures, this speedup technique is called *software pipelining*. Methods of implementing this technique by hand-coding were discussed in [Kogge 77] in the context of pipelined array processors; and a technique for pipelining inner loops without any conditional statements was actually implemented in the FORTRAN compiler for the FPS-164 array processor [Touzeau 84]. David Kuck's group at the University of Illinois have suggested a compilation technique called *dopipe* [Padua 79, Davies 81] to achieve this sort of pipelining on MIMD computers. This technique divides up the loop body to pipeline segments (which are taken to be the maximal strongly connected components or *pi-blocks* of the data dependence graph of the loop body), and allocates each segment to a different processor. Another technique, *doacross* has been proposed by [Cytron 84, Padua 79], which allocates different iterations of the loop to different processors, where the processor containing iteration i starts executing the loop body after a delay proportional to i. Cytron also proved that the problem of deciding whether there exists a semantics-preserving rearrangement of a given loop body so that a given iteration issue delay can be achieved for the loop is NP-complete, which suggests that achieving optimal code with finite resources for this sort of pipelined loops must be computation intensive, although some heuristics have been found, both by [Cytron 84] and also by [Munshi and Simons 87]. But even if we assume that the processors run with the same clock so that the normal synchronization requirements of MIMD machines are alleviated, these iteration pipelining techniques require synchronization at least in the case where a loop iteration can take a variable amount of time to produce a value needed by the next iteration because of conditional statements, and when we do not want to be conservative by taking the worst case time in computing the iteration issue delay. We will suggest a compilation technique for inner loops on VLIW machines, that can perform this sort of iteration pipelining and that avoids the synchronization problem. Our technique works on inner loops that do not contain subroutine calls, but which may contain if-then-else and conditional exit statements. With this technique, called *pipeline scheduling*, a new iteration of an inner loop can be started on every cycle if dependences and resources permit. Given any inner loop body, our algorithm generates the flow graph (actually a multigraph) that represents the possible states of the pipeline in breadth-first fashion, and ensures that the states start repeating without having to generate too many of them.[8] Unlike some previous approaches [e.g. Touzeau 84], which do not allow pauses between the instructions of an iteration (i.e., a "rigid" pipeline) our technique starts a new iteration as early as it can without regard to whether the iteration can finish without pausing, and allows arbitrary pauses between the instructions of an iteration (i.e., a "flexible" pipeline). Also, iterations may complete out of sequence in our technique, e.g., if iteration $n-2$ takes a long path, and iteration n takes a short path.[9] Although the programming details of our

[8] Suppose we call the set consisting of the entry instruction of the generated software-pipelined code, level 1; and we call the successors of the entry instruction that are not in level 1, level 2; and we call the successors of instructions in level 2 that are not in level 2 or 1, level 3, etc.. Then, the maximum number of levels in the generated software-pipelined code is guaranteed not to exceed the length of the longest cycle free path starting at the entry instruction of the original loop. A successor of any instruction at the maximum level is guaranteed to belong to the previous levels or to the same level (unless it is an exit). Note that there is no guarantee that such repeating pipeline states can be obtained within so few levels by unrolling an inner loop a number of times and naively applying trace or list scheduling to the unrolled loop.

[9] There have recently been other techniques that have been independently discovered for software pipelining of loops with tests. The most interesting alternative technique (interesting in terms of competence of the resulting code as viewed by a microprogrammer) seems to be A. Aiken and A. Nicolau's Perfect Pipelining method [Aiken and Nicolau 87], which is applicable to uncompacted loop bodies. The *simple rule* technique that they propose for implementing Perfect Pipelining involves unrolling the loop a number of times, moving up the different iterations in the unrolled code one by one as high up as they can go on all paths, without introducing any pauses between the instructions of an iteration, and then finding redundant microwords (repeating states) in each path starting at the header of the resulting unrolled-compacted code, and removing these redundant microwords and having the edges that went to these removed microwords go to their copies instead. The simple rule does not allow pauses between the instructions of an iteration, i.e it produces a rigid pipeline; which may reduce the iteration issue rate (e.g., iteration n+1 cannot start until iteration n has progressed to a stage where iteration n+1 can start execution and continue without pausing on *all* possible paths that iteration n+1 can take. If pauses were allowed, iteration n+1 could have been able to start earlier and pause on an as-needed basis, and could have allowed the pipelined code to sustain a higher iteration issue rate). However, zero-cycle delays between the issuing of consecutive iterations may be achieved with the Perfect Pipelining method (until resources are exhausted) when some paths of the loop have no inter-

scheduling algorithm are themselves easy to understand, the code generated by the algorithm is very parallel and formidably difficult to understand or imitate by hand-coding even for modest sized loops; so we will just try our best to explain the method clearly. The input of the algorithm is an already compacted inner loop body with a distinguished entry instruction labeled (1), and some other instructions internal to the loop with other numerical labels. The branch target labels that are referred to by instructions in the loop that are not the labels of other instructions within the loop, are called the *exit labels* of the loop, and are of the form (E1), (E2),... (EK). The list of live variables at each exit is also specified with the input. The output of the algorithm is another program which is a software pipelined version of the given loop body.

The algorithm uses a queue of labels which is initially empty. The instructions of the software-pipelined version of the loop are placed in a data structure called schedule, which is also initially an empty list. The entry instruction of the pipeline schedule is a copy of the entry instruction of the loop body, and is also labeled (1), however for each branch target (r) marking the tip nodes of the first instruction of the schedule, where (r) is neither (1) nor an exit, we change (r) to $(r\,(1))$, and enqueue the label $(r\,(1))$ at the end of the label queue, if it is not already in the queue. Then while the label queue is not empty, we repetitively pick a label from the front of the label queue, generate a new instruction of the schedule with that label, and possibly add more labels to the end of the queue during the creation of this new instruction of the schedule.

Consider a label $(p\ q)$ picked up from the front of the queue. Here (p) is always the label of an instruction in the loop body, and q is of the form $(q_1(q_2 \ldots (q_z) \ldots))$, $z \geq 1$, where $(q_1), \ldots, (q_{z-1})$ are labels of instructions within the loop body, and (q_z) is either the label of an instruction within the loop body, or an exit label. Intuitively, (p) is the label of an instruction that belongs to the "current" iteration, and q is either an exit label or the label of a cluster of loop instructions already in the schedule, which belong to "future" iterations.

To construct the instruction labeled $(p\ q)$, we first make a copy of the instruction labeled (p) in the loop body. Then, to each tip node in this copy, we try to append a copy of the instruction labeled q, (which is always already in the schedule, unless q is an exit label), depending on the 5 possible cases:

Let (r) be the target label that marks this tip node of this copy of (p)

```
if (r) is an exit label
    then leave this tip node intact
else if (r) is (1),
    if q is not an exit and q is OK to execute concurrently with this i-branch of (p)**, and resource constraints/heuristics
    are satisfied
        append a copy of q to this tip (by deleting the tip node, and the root node of the copy of q, and connecting to-
        gether the edge that comes in to the tip node and the edge that goes out of the root node of the copy of q).
    else
        change the target label (r) to q in this tip node
    end
else /* (r) is neither (1) nor an exit label */
    if q is not an exit and q is OK to execute concurrently with this i-branch of (p), and resource constraints/heuristics are
    satisfied,
        append a copy of q to this tip. Change each target label s of this copy of q appended to the tip, to (r s), and add
        (r s) to the end of the queue, if it is not already in the queue or schedule.
    else
        change the target label (r) to (r q) in this tip node. Add (r q) to the end of the queue, if it not already in the queue
        or schedule.
    end
end
```

Once all tip nodes in the the copy of the instruction (p) have been modified this way, the modified instruction (p) is added to the schedule, after changing its label to $(p\ q)$. And then, another label is picked from the front of the queue, and another schedule instruction which has that label is generated, etc., and the whole process is repeated until the queue is empty. Pipeline scheduling is formally described in detail, and its termination, and the semantic equivalence of its output to the input loop, are proved in [Ebcioğlu 87].

Often, the only reason that the cluster of instructions q in the schedule belonging to future iterations cannot be executed concurrently with the particular i-branch of the loop body instruction (p) is because q is writing into some register t whose old value will still be used during the current iteration. In this case, if an extra register t'

iteration dependences (In our technique, where the iteration issue delay cannot be less than one cycle, such zero-cycle delays may be achieved not directly, but by unrolling the loop a number of times, compacting the unrolled code, and then applying pipeline scheduling).

** For the case of register dependences only, q is OK to execute concurrently with a particular i-branch of (p), iff q does not write a register that is read or written on a cycle free path in the loop body starting at the target (r) of this i-branch and ending at (1) or an exit; and q does not read a register that is written on this i-branch or on a cycle free path in the loop body starting at the target (r) of this i-branch and ending at (1) or an exit.

is available, and if this i-branch of (p) has no definitions of t, and if an imaginary definition (COPY t t) of t at this tip edge of (p) would cover all of its uses, i.e. no other definitions of t would reach these uses, then these uses of t in the loop body or exits can be renamed as t', and a transfer (COPY t t') (meaning $t \rightarrow t'$) can be added to this tip edge of (p). After these changes, the copy of q can be added harmlessly to this branch of (p), even though q clobbers t. This renaming transformation can be crucial in obtaining a short iteration issue rate; and can be done on-the-fly during the pipeline scheduling algorithm, whenever a pipeline stall is noticed by the algorithm which appears to be fixable by renaming. In the worst case, with infinite resources, the pipeline scheduling technique may generate $O((n-1)!)$ schedule instructions for a given loop of n instructions. But code explosion is prevented to a good extent because of the resource limitations on the size of the microwords.

The appendix gives an example of pipeline scheduling, on a loop that finds the maximum and minimum elements of an array. This loop, taken from [Jensen and Wirth 74] examines two elements of an array in each iteration, and updates the running maximum and running minimum of the array, if they need to be updated. First we give the sequential intermediate code for the inner loop, followed by the same code compacted using greedy percolation scheduling. Finally the compacted loop body is scheduled using pipeline scheduling technique, but assuming infinite resources, in order to demonstrate the available parallelism. As the reader can see, in the steady state part of the loop, the software-pipelined loop throughputs one iteration per cycle except when the minimum or maximum is updated, in which case the pipeline stalls for one cycle. It should be noted that this degree of parallelism is rather good for this sequential-natured algorithm. The technique described above is likely to extract parallelism that is limited only by the number of operation elements. It is superior to loop unrolling [Fisher 82] in serial loops where iteration $n+1$ has some dependency on iteration n, because if such a loop is unrolled k times and then scheduled with an existing technique like trace scheduling, then the k iterations contained within the loop may indeed be pipelined but the initial pipeline filling delays will be incurred on every k iterations. Such delays are incurred only once in the pipeline scheduling technique. Moreover, large numbers of loop unrolling can generate more code than the code generated by our technique.

Proposed machine architecture

The architecture of the proposed machine has been purposefully designed to be streamlined, in the tradition of the 801 and RISC [Patterson 81, Radin 82, Hennessy et al. 82]. However, unlike the single-chip RISC approach, we have decided to make ample use of VLSI for implementing a very high degree of connectivity and easy-to-schedule resources in our machine. We will give a preliminary description of our design ideas below. For architectural features such as the number or registers or ALUs, we will not specify a specific number, but a symbolic constant, since a synchronous computer design is essentially a hard-wired parallel program for interpreting instructions; and we feel that it is not a good idea to use specific numbers in programs, or to proclaim a fixed number for the available amount for an architectural resource and have the software become dependent on this fixed number. We will give after each symbolic constant, in parentheses, what its value will be in the first prototype.

The proposed architecture (which currently does not have a name) consists of N_{ALUS} (8) identical ALUs, each capable of executing a usual repertoire of integer, logical, shifting, field extraction, and floating point operations, as well as conversion operations between integer and floating point. The floating point format is IEEE single precision, and the integer format is 32 bit two's complement and 32 bit unsigned. However, every operation yields a result: There is a special bit pattern $\perp$, called "bottom" (obtained via an additional exception tag bit - a 33rd bit) analogous to the bottom element in lattice theory or in Backus's FP language, that is the result of operations that would normally cause an exception. If an operation would normally cause an exception (such as integer overflow), or if one of the operands of the operation is $\perp$, the result is $\perp$. There are interruptible (exception-raising) and uninterruptible versions of each operation. The interruptible version causes an exception when the result is $\perp$, the uninterruptible version never causes an exception even when the result is $\perp$. This type of architecture is required in order to aggressively execute operations ahead of time, without fear of incurring e.g., an overflow or division by zero exception which would not have happened in the sequential version of the program.

The main communication scheme between the ALUs is a $3N_{ALUS}$ (24) port register file consisting of N_{REGS} (64) 33-bit registers (counting the exception tag bit). At the beginning of every clock period a given ALU can read any two registers as input and can transfer its result to any register (conceptually) at the end of the same clock period. If more than one ALU result is simultaneously written into a register, the written value is architecturally undefined (however, the values are or'ed together in the present implementation proposal). RISC-like pipelining techniques with bypass paths (similar to those described in [Agerwala and Cocke 87]) are used to reduce cycle time.[11] Non-pipelined versions of multiport register files with this organization (but with a small number

[11] The result of an ALU goes into a dedicated ALU output register at the end of the current cycle, and is stored into the destination register on the next cycle. The data is taken from the dedicated register if another ALU tries to read the new value of the destination register during the next cycle. This involves too many bypass paths, but register file design is pin-limited, not area limited

of ports) were previously used in the Nanodata QM-1 [Nanodata 79] and the Kyoto university QA-1 and QA-2 machines [Hagiwara et al. 80, Tomita et al. 86].

The ALUs are combinatorial: integer addition, subtraction, logical operation and shifting can be done in a single cycle. For operations that cannot be done in a single cycle, the inputs must be held constant and the result will be valid after a predetermined number of cycles. A multiple-cycle ALU operation can be spread out over several consecutive instructions as long as the inputs of the ALU are kept constant; the other ALUs can perform many single-cycle operations during this time. Sequential operations such as division also appear combinatorial (e.g., as implemented in the BIT ALU), and their inputs must also be held constant throughout the entire operation. The reason we chose combinatorial, rather than pipelined floating point operations is because they simplify the design, they are more amenable to precise interrupts, and they tend to provide faster execution in inherently sequential code, e.g. pipelined loops where iteration n+1 depends on a result from iteration n.

The data memory address is 32 bits long and indicates a byte address. A store operation can write 32-bit fullwords, aligned 16-bit halfwords, or 8-bit bytes. A read operation can only read a fullword, which can be broken apart by a subsequent operation. Fullword memory operations can also be done uninterruptibly, in which case a read of a non-existent or protected location will return the special $\perp$ value, and a store operation will store $\perp$ in memory without causing an exception. Memory access is achieved by $N_{ALUS}/2$ (4) alternating address and data ports to memory. The left input of ALU 0 and the output of ALU 1 constitute the address and data paths of memory port 0. Memory ports 1,2,3,..., $(N_{ALUS}/2) - 1$ are similarly defined for the remaining pairs of ALUs. There are N_{BANKS} (8) independent banks in the prototype memory, such that consecutive words are in consecutive banks. Memory operations take a single cycle in the absence of bank conflicts. When there is a request from more than one port for a given bank, the requests are satisfied in increasing port order, and the instruction clock is held until all requests are satisfied. We felt that such hardware assist had to be provided since memory bank disambiguation at compile time is less successful for non-numerical code than it is for parallel scientific code. Note that the data to be stored is taken from the output of an ALU, so there is no time to really store it in the same cycle. A buffering technique is used instead, which nevertheless allows the new value of the memory location to be read on the cycle immediately following the "store," via a bypass path from the buffer register.

The instruction word is 537 bits wide. For each ALU an operation, two sources and a destination is specified. Each input of an ALU can be taken from either a register or from one of the N_{IMM} (6) 32-bit immediate fields in the microword (some of the same immediate fields can also be read as sign extended 16-bit fields). The destination is specified as a register number, and a $N_{TARGETS}$ (4) bit transfer enable mask. (The machine can do a multiway branch to $N_{TARGETS}$ target instructions). Suppose that at the beginning of an instruction, the current values of the condition code registers are such that the instruction branches to target i, $0 \le i \le N_{TARGETS} - 1$. Then, the transfer to the destination register of this ALU takes place only if i'th bit of the transfer enable mask for this ALU is 1. The purpose of this mechanism is to allow the conditional execution mechanism of the decision tree shaped instructions described above. For example, assume that an ALU operation occurs (in a non-overridden position) on the first two i-branches of the instruction-tree; but not in the remaining i-branches. The compiler will then set bits 0 and 1 of the operation's transfer enable mask to 1, and will set the remaining bits of its transfer enable mask to 0. In addition to the ALU parcels and immediate fields in the microword, there also are memory port parcels for each memory port, and branch control masks for conditional branching.

The conditional branch mechanism was designed to support the if-then-else tree multiway branching mechanism, and to allow nonsense comparisons (such as comparing the result of a division by zero to an integer) to be executed ahead of time, as long as they do not affect a subsequent branching decision. There are N_{ALUS} 2-bit condition code registers, one for each ALU. Comparison operations (such as GT, LT, EQ for two's complement comparisons, and their unsigned and floating point variants) set the condition code register for the particular ALU. The possible values for a condition code register are true (01), false (00), and error (1X). Comparisons that involve the special $\perp$ value as an operand set the error condition code, and also cause an exception if done interruptibly. Three valued logic (true, false, error) is used for next address selection.[12] For each branch target i, $0 \le i < N_{TARGETS} - 1$, there is a pair of N_{ALUS}-bit masks in the microword, A_i, and B_i, which specify two arbitrary subsets of the condition code (cc) registers. At the beginning of the instruction, for each branch target i, $0 \le i < N_{TARGETS} - 1$, a three-valued logic signal called $target_i$ is computed as follows: If A_i and B_i are disjoint, and each cc in A_i is true and each cc in B_i is false, then $target_i$ is true, otherwise if A_i and B_i are not disjoint, or at least one cc in A_i is false or at least one cc in B_i is true, then $target_i$ is false, otherwise, $target_i$ is error. After computing these signals, the machine finds the first i such that $target_i$ is not false (it better be true, otherwise a branching error exception occurs); and then selects the absolute address of the next instruction from immediate field #i. But if all target signals are false, then the next instruction address is taken to be the address

[12] with the relevant truth table entries being (F and E)=F, (T and E)=E, (E and E)=E, (T or E)=T, (F or E)=E, (E or E)=E, (not E)=E. It is interesting to note that [Kleene 52] has used a three-valued logic with the same truth tables as the ones used here, for representing the values of computable predicates, with the third truth value reserved for the case where the program implementing the predicate does not terminate.

of the next sequential instruction, which is considered to be target number $N_{TARGETS} - 1$. All of this is to enable the parallel execution of Boolean expressions such as (y!=0 && x/y>3) in C, where x/y>3 can be computed without having to wait for y!=0 to complete, and where the condition codes resulting from y!=0 and x/y>3 can be tested simultaneously without incurring a branching error exception, even if y is indeed 0.[13] Unconditional branch is obtained by placing an always true pattern (null sets) in the first pair of masks, no-branch is achieved by placing an always false pattern (intersecting sets) in all the pairs of masks. In addition to the $N_{TARGETS}$ possible targets, an additional escape pattern in the masks forces the next address to be taken from the output of ALU 1. This is necessary for performing computed goto, returning from subroutines, and calling functions passed as parameters.

Protection is provided through supervisor and user modes. When an interrupt is detected, control branches to location 0+some offset determined by the type of interrupt. Interrupts are disabled and processor is forced to supervisor mode. If an interrupt logically occurs during instruction n, it is detected at the beginning of instruction n+1, and the updating of the register file by instruction n is inhibited (remember that the register file is updated late, due to pipelining), and the privileged "oldpsw" register is loaded with the address of instruction n, plus the condition codes and other process state registers as they existed in the beginning of instruction n.[14] All I/O is memory mapped, and I/O accesses are possible only in supervisor mode. A return from an interrupt (load PSW) is accomplished in the supervisor mode by a special variant of the branch-to-ALU-1 instruction that also loads the condition code registers and other process state registers from the output of ALU-3 (and of ALU-5, ALU-7,... as determined by the PSW length implied by the architecture parameters), and stretches the clock during the next instruction if necessary, so that any interrupted long combinatorial operations are started from scratch and are given ample time to complete.

Note that the architecture described so far does not define any virtual memory mechanism, cache, or I/O system, but clearly allows precise interrupts, and provides the primitives so that some memory hierarchy can be designed. To make the project realizable, we wish to decouple the design of the caches, virtual memory mechanism, and I/O system from the CPU design. At this point we only want to study the behavior of the CPU with infinite cache performance. So we will only build a version with a small amount of fast memory (16K words (~1M bytes) of instruction memory and 128K words (512K bytes) of data memory), which will be attached to an IBM PC/AT through an interface card. This interface card will be able to start, stop, and single-step the clock of the VLIW machine, read and write all of its internal registers through a scan ring, read and write its memories, and exchange interrupts with it. This way we will be able to download and run some small benchmarks in the VLIW machine. As for the software side, we plan to develop a compactor that takes the intermediate code output from the C version of the PL.8 compiler [Warren et. al 86], and the assembly language output from the FORTVS2 Fortran compiler, re-performs certain traditional optimizations to compensate for architectural differences between the IBM 370 and the VLIW machine, and then applies percolation and pipeline scheduling to obtain compacted VLIW machine code.

For the prototype, we are planning to use LSI Logic CMOS compacted arrays with high I/O pin count, commercially available ALUs from Bipolar Integrated Technology, and 17ns 16K*4 CMOS static RAMs. The 24-port register file will be implemented as two-bit slices of the entire register file per chip (17 copies will be used), and will also include the immediate field multiplexers for the ALU inputs. Another chip, the next address multiplexer (4 copies), will incorporate the irregular logic of the CPU involving interrupts and the PSW, as well as the next instruction address multiplexer. A third chip, the memory crossbar switch (8 copies), will handle the traffic between memory ports and banks and arbitration. Presently, for the path: register file access → ALU operation → dedicated register setup; and the path: compute and drive next address → instruction memory RAM access time → instruction register setup; designing for a 50ns cycle time *looks* possible. The path: register file access → memory crossbar switch → RAM access time → crossbar switch → dedicated register setup is slower, because of two extra chip crossings; and we are considering alternative organizations (such as making true multiport RAMs that can later be used as a building block in a direct mapped cache) to remedy this. Thus, assuming we count the actions of each of the 8 ALUs, each of the 4 memory accesses, and each of the 3 conditional branches (4 branch targets = 3 test nodes in the if-then-else tree) as the equivalent of a RISC instruction, the architecture will be able to execute a maximum of 15 RISC instructions per cycle. However, we feel that it is not very wise to proclaim peak MIPS ratings, especially for nonexistent machines.

---

[13] Note that in the sequential semantics of C, first y!=0 is computed within (y!=0 && x/y>3), and if y!=0 is false, x/y>3 is not computed.

[14] This technique obviously helps to reduce the cycle time. Also, a multiport direct mapped cache can use this mechanism to send possibly wrong data to the CPU during instruction n without checking the cache directory contents, and then signaling a delayed-interrupt at the beginning of instruction n+1 which will restore the machine state as it existed at the beginning of instruction n. The memory mechanism will have to arrange that the fetching of the interrupt routine instruction (which will just return from the interrupt) does not cause a cache miss. Upon return from interrupt, instruction n will be re-executed, and the cache miss mechanism can then supply correct data for all requests from the memory ports, holding the clock until the cache miss processing is complete, if necessary. This can allow very short cache access times when there is no cache miss. But so far we have no immediate plans for making a cache.

Preliminary performance predictions

To get a rough estimate of the performance of the proposed machine, we hand-compiled the less complex Lawrence Livermore Loops benchmarks for the proposed machine, using greedy percolation scheduling that takes operations in the order of appearance in the program and moves them up on all paths as far as they can go, followed by pipeline scheduling, which we feel we can implement in a VLIW compiler.[15] Since these loops are short, it is possible to calculate the execution time by hand. We have assumed that floating point add, subtract and multiply, and integer multiply, have been combinatorially implemented and take 2 cycles each (100ns for a 50ns clock - this is a conservative assumption); and that the slowest operation in a microword just stalls the other operations. We have also hand-compiled 4 (embarrassingly) simple C programs to estimate performance on very sequential-natured software: a program to sum the elements of an integer array, an insertion sort program that sorts ten integers, a merging program that takes two ten element sorted integer arrays and merges them into a third 20-element integer array, and a recursive factorial program that finds 10!. We will first present some comparisons of the performance of the machine against a similarly constructed RISC which can execute only one three-register operation, or one memory load or store, or one conditional or unconditional branch in a single cycle (but floating point operations also take 2 cycles on the RISC machine). We will compare the number of cycles required to finish a program segment on this RISC and on the VLIW in order to get an abstract measure of inherent parallelism that can be captured by these compilation techniques. In the table below, the number of cycles in the inner loop of each program is given, except for kernel 21 (matrix multiply for 25*25 matrices), insertion sort, and recursive factorial, where the number of cycles for the entire program is given. As we can see, percolation scheduling, as modified by us, already gives a speedup of about 3.0. The addition of the pipeline scheduling on top of percolation scheduling gives a parallelism of about 4.9 and up to 6.7 in some sequential-natured programs. Notice that more traditional parallelism extraction techniques such as loop unrolling or recurrence breaking (e.g. for the inner product computation in Kernel 3) have not been used to obtain these results (such techniques could yield additional parallelism).

prog	RISC cycles	VLIW cycles compac.	VLIW cycles pipel.	speedup compac.	speedup pipel.
kernel1	21	9	4	2.33	5.25
kernel3	11	5	2	2.20	5.50
kernel5	13	5	4	2.60	3.25
kernel11	9	3	2	3.00	4.50
kernel21	197052	82627	35752	2.38	5.51
kernel24	10	4	3	2.50	3.33
ins. sort	497	173	74	2.87	6.72
merge	15.5	3	3	5.17	5.17
rec. factorial	235	65	65	3.62	3.62
arraysum	6	2	1	3.00	6.00
average				2.97	4.89

In the appendix, we give an example of one of the programs used here: the merge program, its RISC assembly code version, and its VLIW machine code version. Pipeline scheduling does not work for this sequential merging algorithm because the first instruction of iteration n+1 uses a register set by the last instruction of iteration n; nevertheless, ordinary percolation scheduling is able to achieve some speedup. It should be noted that although the speedup is about 5 on this program, an average of about 7 operations/conditional jumps are being executed in each instruction of the program: it seems that some extra operations/conditional jumps that belong to *untaken* paths have to be executed ahead of time or conditionally, to achieve speedup on this kind of non-numerical code.

Now, a well designed RISC will have a cycle time that is somewhat shorter in a given technology (for example, the VLIW cycle time would be about 30% longer than the bypass path of a four stage pipelined RISC in AS-TTL), and the RISC could overlap branches with arithmetic operations so that unconditional branches take zero time [Ditzel and McLellan 87]. So this speedup value must be derated for a realistic comparison. But it is difficult to make a comparison to a hypothetical machine. Since future RISC's will undoubtedly approach supercomputer speeds, we have instead collected some statistics that compare the VLIW architecture to a real

---

[15] We are past the hand-compiling stage at present. We presently have a working preliminary version of the compiler back-end (coded by Mauricio Breternitz) that performs percolation and pipeline scheduling (including the renaming optimization), gate level schematics for the prototype, and a cycle-simulator written in C, that accurately models every signal, bus and flip-flop in the machine. We will therefore report more accurate predictions about the performance of our machine in future papers.

ed machine. we hand-compiled the less complex
achine, using greedy percolation scheduling that
nd moves them up on all paths as far as they can
plement in a VLIW compiler.[15] Since these loops
d. We have assumed that floating point add, sub-
rially implemented and take 2 cycles each (100ns
at the slowest operation in a microword just stalls
arrassingly) simple C programs to estimate per-
sum the elements of an integer array, an insertion
takes two ten element sorted integer arrays and
rsive factorial program that finds 10!. We will first
e against a similarly constructed RISC which can
oad or store, or one conditional or unconditional
ke 2 cycles on the RISC machine). We will com-
ent on this RISC and on the VLIW in order to get
ired by these compilation techniques. In the table
am is given, except for kernel 21 (matrix multiply
where the number of cycles for the entire program
by us, already gives a speedup of about 3.0. The
eduling gives a parallelism of about 4.9 and up to
traditional parallelism extraction techniques such
oduct computation in Kernel 3) have not been used
l parallelism).

	speedup compac.	speedup pipel
	2.33	5.25
	2.20	5.50
	2.60	3.25
	3.00	4.50
	2.38	5.51
	2.50	3.33
	2.87	6.72
	5.17	5.17
	3.62	3.62
	3.00	6.00
	2.97	4.89

s used here: the merge program, its RISC assembly
scheduling does not work for this sequential merg-
ses a register set by the last instruction of iteration
achieve some speedup. It should be noted that al-
of about 7 operations/conditional jumps are being
me extra operations/conditional jumps that belong
ditionally, to achieve speedup on this kind of non-

newhat shorter in a given technology (for example,
he bypass path of a four stage pipelined RISC in
netic operations so that unconditional branches take
must be derated for a realistic comparison. But it
e. Since future RISC's will undoubtedly approach
istics that compare the VLIW architecture to a real

a working preliminary version of the compiler back-end (coded
eduling (including the renaming optimization), gate level sche-
accurately models every signal, bus and flip-flop in the machine.
mance of our machine in future papers.

supercomputer, the 3090.[16] Listed below are the execution times in ns for each inner loop iteration (complete program execution time for kernel 21, insertion sort, and recursive factorial) for the IBM 3090 and the proposed VLIW machine (in terms of the VLIW cycle time c). On the 3090 the Fortvs2 Fortran compiler, and the C version of the PL.8 compiler were used, with all optimizations and vectorizations turned on. Both of these compilers are probably the best optimizing compilers available on the architecture for these languages. Note that since the programs access little data, and were timed by executing them millions of times in a loop and then reading the virtual cpu time, we can reasonably assume that we are comparing the infinite cache performance of the 3090 against the infinite cache performance of the VLIW. There is a wide variation in the performance ratio (for example the VLIW requires a cycle time of 18ns to catch up with the 3090 on the vectorized matrix multiply kernel, but 278ns on the merging program), mainly because of some vectorizable loops where the 3090 is very fast. The finite cache performance of this VLIW machine will of course depend critically on the sizes of the instruction and data caches and the line miss mechanism. But large caches are quite possible to implement, for virtual memory systems designed from scratch, if the cache is made visible to the operating system software.

prog.	VLIW time (ns)	3090 time (ns)	ratio
kernel1	4c	142(V)	36/c
kernel3	2c	82(V)	41/c
kernel5	4c	242	61/c
kernel11	2c	223	112/c
kernel21	35752c	650000(V)	18/c
kernel24	3c	260	87/c
ins. sort	74c	14200	192/c
merge	3c	834	278/c
rec. factorial	65c	11000	169/c
arraysum	1c	150	150/c
average			114/c

(V) = vectorized loop

Conclusions

We have described our current progress with a VLIW architecture intended for parallel execution of sequential, non-numerical code as well as scientific code. The degree of success of VLIW architectures intended for executing parallel scientific code is more or less known at this stage. The present proposal is a research experiment to probe the effectiveness of VLIW architectures and compilation techniques for sequential-natured software. Much work still has to be done to investigate the usefulness of the ideas presented herein, and it is too early to jump to hard conclusions; but VLIW machines seem to be a promising research area to pursue in computer architecture, and seem to offer novel possibilities for speeding up inherently sequential code, which cannot be adequately speeded up by multiprocessors or vector supercomputers. An effort is now underway at the IBM Thomas J. Watson Research Center to build a prototype of this machine, and we will report on our progress in future papers.

Acknowledgements

I am grateful to Fran Allen, Mauricio Breternitz, Michael Burke, John Cocke, Ron Cytron, Monty Denneau, Dave George, Manoj Kumar, and George Radin for their helpful comments on the architecture and compilation techniques described in this paper.

References

Adam, T.L., Chandy, K.M., and Dickson, J.R. (74) "A Comparison of List Schedules for Parallel Processing Systems" Communications of the ACM 17, 12, December 1974.

Agerwala, T. (76) "Microprogram Optimization: A Survey" IEEE Transactions on Computers 25, October 1976.

Agerwala, T. and Cocke, J. (87) "High Performance Reduced Instruction Set Computers" research report no. RC 12434, IBM Thomas J. Watson Research Center, Yorktown Heights, 1987.

* These are my own informal measurements, and should definitely not be construed as any official indication of the performance of an IBM product.

Aiken, A. and Nicolau, A. (87) "Perfect Pipelining: A New Loop Parallelization Technique" TR 87-873, Dept. of Computer Science, Cornell University, October 1987.

Allen, R.A. and Kennedy, K. (84) "Automatic Translation of Fortran Programs to Vector Form" Rice Technical Report No. TR84-9, Dept. of Computer Science, Rice University, July 1984.

Arnould, E., Kung, H.T., Menzilcioglu, O., Sarocky, K., (85) "A Systolic Array Computer" Proc. of the 1985 International Conference on Acoustics, Speech, and Signal Processing (March 1985).

Arvind, and Ianucci, R.A. (83) "A Critique of Multiprocessing von Neumann Style" Proc. 10th Annual International Conference on Computer Architecture, 1983.

Beetem, J., Denneau, M., and Weingarten, D. (85) "The GF11 Supercomputer" The 12th Annual International Symposium on Computer Architecture, June 1985.

Burke, G.R. (87) "A Multiport Register File Chip for the CHoPP Supercomputer" VLSI Systems Design, August 1987.

Charlesworth, A.E. (81) "An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family" IEEE Computer, September 1981.

Colwell, R.P., Nix, R.P., O'Donnell, J.J., Papworth, D.B., and Rodman, P.K. "A VLIW Architecture for a Trace Scheduling Compiler" Proc. ASPLOS 1987.

Cydrome Inc. (88), "Cydra 5 Directed Dataflow Architecture: Summary" Milpitas, California, 1988.

Cytron, R.G. (84) "Compile-time Scheduling and Optimization for Asynchronous Machines" Report no. UIUCDCS-R-84-1177, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1984.

Davies, J.R.B.(81) "Parallel Loop Constructs For Multiprocessors" Report no. UIUDCS-R-81-1070, Dept. of Computer Science, University of Illinois at Urbana-Champaign, May 1981.

Dennis, J.B. (74) "First Version of a Data Flow Language" Proceedings, Colloque sur la Programmation, Lecture Notes in Computer Science 19, April 1974.

Dennis, J.B. (80) "Data Flow Supercomputers" Computer 13(11), November 1980.

Ditzel, D.R., and McLellan, H.R. (87) "Branch Folding in the CRISP microprocessor: Reducing Branch Delay to Zero" Proceedings of the 14th Annual International Symposium on Computer Architecture, June 1987.

Ebcioğlu, K. (87) "A Compilation Technique for Software Pipelining of Loops with Conditional Jumps" Proc. MICRO-20, ACM Press, December 1987.

Fisher, J. A. (79) "The Optimization of Horizontal Microcode within and beyond Basic Blocks: An Application of Processor Scheduling with Resources" Ph.D. Thesis, Dept. of Computer Science, New York University, October 1979.

Fisher, J.A. (83) "Very Long Instruction Word Architectures and the ELI-512" Proc. 10th Annual Symposium on Computer Architecture, June 1983.

Fisher, J.A. and O'Donnell, J.J. (84) "VLIW Machines: Multiprocessors We Can Actually Program" Proc. Compcon 84, February 1984.

Flynn, M.J. (66) "Very High Speed Computer Systems" Proc. of the IEEE, Vol. 54, No. 12, December 1966.

Foster, C.C., and Riseman, M.R. (72) "Percolation of Code to Enhance Parallel Dispatching and Execution" IEEE Transactions on Computers, December 1972.

Gajski, D.D., Padua, D.A., Kuck, D.J., Kuhn, R.H. (85) "A Second Opinion on Data Flow Machines and Architectures" IEEE Computer, Vol. 15, No. 2, February 1982.

Goodman, J.R., Hsieh, J.T., Liou, K., Pleszkun, A.R., Schechter, P.B., Young, H.C. (85), "PIPE: A VLSI Decoupled Architecture" The 12th Annual International Symposium on Computer Architecture, June 1985.

Hagiwara, H., Tomita, S., Oyanagi, S., Shibayama, K. (80) "A Dynamically Microprogrammable Computer with Low-level Parallelism" IEEE Transactions on Computers, Vol C-29, no. 7, July 1980.

Hennessy et al. (82) "The MIPS Machine" Digest of Papers - Compcon Spring 82, February 1982.

Jensen, K. and Wirth, N. (74) "Pascal User Manual and Report" Springer-Verlag, 1974.

Karplus, K., and Nicolau, A. (85), "Efficient Hardware for Multi-way Branches and Pre-fetches" Proc. of the 18th Annual Workshop on Microprogramming, 1985.

Kleene, S.C. (52) "Introduction to Metamathematics" Van Nostrand and Company, 1952.

Kogge, P.M. (77) "The Microprogramming of Pipelined Processors" Fourth Annual Symposium on Computer Architecture, 1977.

Kuck, D.J. (78) "The Structure of Computers and Computations" Vol. 1, John Wiley and Sons, 1978.

Lah, J. and Atkins, D.E. (83) "Tree compaction of Microprograms" Proc. 16th Annual Microprogramming Workshop, October 1983.

Lee, G., Kruskal, C.P., and Kuck, D.J. (85) "The Effectiveness of Automatic Structuring on Nonnumerical Programs" Proc. 1985 International Conference on Parallel Processing, 1985.

Machtey, M., and Young, P. (78) "An Introduction to the General Theory of Algorithms" Academic Press, 1978.

Munshi, A.A., and Simons, B. (87) "Scheduling Loops on Processors: Algorithms and Complexity" Research report no. RJ 5546, IBM Thomas J. Watson Research Center, Yorktown Heights, March 1987.

Nanodata Computer Corporation (79) "QM-1 Hardware Level User's Manual" Buffalo, New York, 1979.

›op Parallelization Technique" TR 87-873, Dept.

⁝ Fortran Programs to Vector Form" Rice Tech-
·e University, July 1984.
"A Systolic Array Computer" Proc. of the 1985
l Processing (March 1985).
ıg von Neumann Style" Proc. 10th Annual Inter-

l Supercomputer" The 12th Annual International

oPP Supercomputer" VLSI Systems Design, Au-

· Processing: The Architectural Design of the
1981.
nd Rodman, P.K. "A VLIW Architecture for a

: Summary" Milpitas, California, 1988.
ation for Asynchronous Machines" Report no.
ersity of Illinois at Urbana-Champaign, October

ssors" Report no. UIUDCS-R-81-1070, Dept. of
paign, May 1981.
" Proceedings, Colloque sur la Programmation,

(11), November 1980.
CRISP microprocessor: Reducing Branch Delay
mposium on Computer Architecture, June 1987.
pelining of Loops with Conditional Jumps" Proc.

: within and beyond Basic Blocks: An Application
·ept. of Computer Science, New York University,

and the ELI-512" Proc. 10th Annual Symposium

ultiprocessors We Can Actually Program" Proc.

. of the IEEE, Vol. 54, No. 12, December 1966.
to Enhance Parallel Dispatching and Execution"

A Second Opinion on Data Flow Machines and
1982.
ter, P.B., Young, H.C. (85), "PIPE: A VLSI De-
nposium on Computer Architecture, June 1985.
"A Dynamically Microprogrammable Computer
uters, Vol C-29, no. 7, July 1980.
Compcon Spring 82, February 1982.
ırt" Springer-Verlag, 1974.
Multi-way Branches and Pre-fetches" Proc. of the

ostrand and Company, 1952.
essors" Fourth Annual Symposium on Computer

ons" Vol. 1, John Wiley and Sons, 1978.
rograms" Proc. 16th Annual Microprogramming

ness of Automatic Structuring on Nonnumerical
lel Processing, 1985.
General Theory of Algorithms" Academic Press,

ocessors: Algorithms and Complexity" Research
nter, Yorktown Heights, March 1987.
vel User's Manual" Buffalo, New York, 1979.

Nicolau, A. (85) "Percolation Scheduling: A Parallel Compilation Technique" TR 85-678, Dept. of Computer Science, Cornell University, May 1985.

Padua-Haiek, D.A. (79) "Multiprocessors: Discussion of Some Theoretical and Practical Problems" Report no. UIUCDCS-R-79-990, University of Illinois at Urbana-Champaign, November 1979.

Patterson, D.A., et al. (81) "RISC-I: A Reduced Instruction Set VLSI Computer" Eighth Annual Symposium in Computer Architecture, May 1981.

Rau, B.R., Glaeser, C.D. (81) "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High-performance Scientific Computing" Proc. 14th Annual Microprogramming Workshop, October 1981.

Rau, B.R., Glaeser, C.D., and Picard, R.L. (82) "Efficient Code Generation for Horizontal Architectures: Compiler Techniques and Architectural Support" Proc. 9th Symposium on Computer Architecture, April 1982.

Radin, G. (82) "The 801 Minicomputer" Proc. ACM Symposium on Architecture Support for Programming Languages and Operating Systems, March 1982.

Rogers, H. (67) "Theory of Recursive Functions and Effective Computability" Prentice-Hall, 1967.

Russell, R.M. (78) "The Cray-1 Computer System" Communications of the ACM, vol. 21, no. 1, January 1978.

Thornton, R.E. (64) "Parallel Operation in the Control Data 6600" AFIPS Proc. FJCC, pt. 2, vol. 26, 1964.

Tomasulo, R.M. (67) "An Efficient Algorithm for Exploiting Multiple Arithmetic Units" IBM Journal of Research and Development, vol. 11, January 1967.

Tomita, S., Shibayama, K., Toshiyuki, N., Yuasa, S., and Hagiwara, H. (86) "A Computer with Low-Level Parallelism QA-2" Proc. 13th Annual International Symposium on Computer Architecture, 1986.

Veidenbaum, A. (85) "Compiler Optimizations and Architecture Design Issues for Multiprocessors" Ph.D. thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, May 1985.

Warren, S.H., Auslander, M.A., Chaitin, G.J., Chibib, A.C., Hopkins, M.E., and MacKay, A.L. (86) "Final Code Generation in the PL.8 Compiler" research report RC11974, IBM Thomas J. Watson Research Center, 1986.

Yau, S.S., Schowe, A.C., and Tsuchiya, M. (74) "On Storage Optimization of Horizontal Microprograms" MICRO-7, Sept. 1974.

APPENDIX: CODE EXAMPLES

PIPELINE SCHEDULING EXAMPLE

```
%% From Jensen and Wirth 74, Pascal User Manual and Report, p. 37
%% find the largest and the smallest number in a given list
%% ...
%% min:= a[1]; max:= min; i:= 2;
%% while i < n do
%% begin u:=a[i]; v:=a[i+1];
%%     if u>v then
%%     begin if u>max then max:=u;
%%         if v<min then min:=v
%%     end else
%%     begin if v>max then max:= v;
%%         if u<min then min:= u
%%     end ;
%%     i:= i+2;
%% end
%% ...
```

Three address code for inner loop

```
LOOP
   (LT AI AILIM CC0)
   (IF (NOT CC0) (GOTO EXIT))
   (LOAD A AI U)
   (LOAD A AI1 V)
   (GT U V CC1)
   (IF (NOT CC1) (GOTO L1))
   (GT U MAX CC2)
   (IF (NOT CC2) (GOTO L2))
   (COPY U MAX)
L2
   (LT V MIN CC3)
   (IF (NOT CC3) (GOTO L5))
   (COPY V MIN)
   (GOTO L5)
L1
   (GT V MAX CC4)
   (IF (NOT CC4) (GOTO L3))
   (COPY V MAX)
L3
   (LT U MIN CC5)
   (IF (NOT CC5) (GOTO L5))
   (COPY U MIN)
L5
   (ADD AI 8 AI)
   (ADD AI1 8 AI1)
   (GOTO LOOP)
```

Percolation scheduling result:

The percolation scheduling result already reduces the loop to three cycles: it performs the loads from memory of the two array elements and compares the loop index to the loop bound in the first instruction (1), then it performs the five comparisons involving these two array elements and min and max in the second instruction (2), and then it finally performs the conditional update of min or max in the third instruction (3).

```
(1):
   ((LT AI AILIM CC0) (LOAD A AI U) (LOAD A AI1 V) (ADD AI 8 AI) (ADD AI1 8 AI1)
    (GOTO (2)))
(2):
   ((GT U V CC1) (GT U MAX CC2) (LT V MIN CC3) (GT V MAX CC4) (LT U MIN CC5)
    (IF (NOT CC0) ((GOTO (E1))) ELSE ((COPY U U_P) (COPY V V_P) (GOTO (3)))))
(3):
   ((IF
     (NOT CC1)
     ((IF
```

```
        (NOT CC4)
        ((IF (NOT CC5) ((GOTO (1))) ELSE ((COPY U__P MIN) (GOTO (1)))))
        ELSE
        ( (COPY V__P MAX)
         (IF (NOT CC5) ((GOTO (1))) ELSE ((COPY U__P MIN) (GOTO (1)))) ))
        )
      ELSE
      ( (IF
        (NOT CC2)
        ((IF (NOT CC3) ((GOTO (1))) ELSE ((COPY V__P MIN) (GOTO (1)))))
        ELSE
        ( (COPY U__P MAX)
         (IF (NOT CC3) ((GOTO (1))) ELSE ((COPY V__P MIN) (GOTO (1)))) ))
        )))
(E1):
 ...
```

Pipeline scheduling result:

The pipeline scheduling result executes instruction (1) of the first iteration during its first instruction, and instructions (2) and (1) of iterations 1 and 2, respectively, during its second instruction (2 (1)). The third pipeline schedule instruction (3 (2 (1)) is entered in state where iterations n, n+1, n+2 are expecting to execute instructions (3), (2), (1) respectively (this *expectation state* is notated as n:(3), n+1:(2), n+2:(1)). If (3) of iteration n does not update min or max, all of the expected instructions are executed and a branch is taken back to (3 (2 (1))) with the expectation state n+1:(3), n+2:(2), n+3:(1); otherwise, iterations n+1 and n+2 wait (since, e.g., instruction (2) of iteration n+1 needs the new values of min and max to do its comparisons), and iteration n executes (3) alone, and a branch is taken to (2 (1)) with the expectation state n+1:(2), n+2:(1). (2 (1)) will then branch back to (3 (2 (1))) with expectation state n+1:(3), n+2:(2), n+3:(1), always assuming that the loop is not exited.

```
(1):
  ( (LT AI AILIM CC0) (LOAD A AI U) (LOAD A AI1 V) (ADD AI 8 AI) (ADD AI1 8 AI1)
   (GOTO (2 (1))) )

(2 (1)):
  ( (GT U V CC1) (GT U MAX CC2) (LT V MIN CC3) (GT V MAX CC4) (LT U MIN CC5)
   (IF
    (NOT CC0)
    ((GOTO (E1)))
    ELSE
    ( (COPY U U__P) (COPY V V__P) (LT AI AILIM CC0) (LOAD A AI U) (LOAD A AI1 V)
     (ADD AI 8 AI) (ADD AI1 8 AI1) (GOTO (3 (2 (1)))) )) )

(3 (2 (1))):
  ( (IF
    (NOT CC1)
    ( (IF
      (NOT CC4)
      ( (IF
        (NOT CC5)
        ( (GT U V CC1) (GT U MAX CC2) (LT V MIN CC3) (GT V MAX CC4)
         (LT U MIN CC5)
         (IF
          (NOT CC0)
          ((GOTO (E1)))
          ELSE
          ( (COPY U U__P) (COPY V V__P) (LT AI AILIM CC0) (LOAD A AI U)
           (LOAD A AI1 V) (ADD AI 8 AI) (ADD AI1 8 AI1)
           (GOTO (3 (2 (1)))) )) )
        ELSE
        ((COPY U__P MIN) (GOTO (2 (1))))) )
      ELSE
      ( (COPY V__P MAX)
       (IF
        (NOT CC5)
        ((GOTO (2 (1))))
        ELSE
        ((COPY U__P MIN) (GOTO (2 (1))))) )) )
    ELSE
    ( (IF
```

37

cles: it performs the loads from memory of the two
the first instruction (1), then it performs the five
the second instruction (2), and then it finally per-
3).

D AI 8 AI) (ADD AI1 8 AI1)

MAX CC4) (LT U MIN CC5)
PY V V__P) (GOTO (3)))))

```
        (NOT CC2)
        ( (IF
          (NOT CC3)
          ( (GT U V CC1) (GT U MAX CC2) (LT V MIN CC3) (GT V MAX CC4)
           (LT U MIN CC5)
           (IF
            (NOT CC0)
            ((GOTO (E1)))
            ELSE
            ( (COPY U U__P) (COPY V V__P) (LT AI AILIM CC0) (LOAD A AI U)
             (LOAD A AI1 V) (ADD AI 8 AI) (ADD AI1 8 AI1)
             (GOTO (3 (2 (1)))) )) )
          ELSE
          ((COPY V__P MIN) (GOTO (2 (1))))) )
        ELSE
        ( (COPY U__P MAX)
         (IF
          (NOT CC3)
          ((GOTO (2 (1))))
          ELSE
          ((COPY V__P MIN) (GOTO (2 (1))))) )) )) )
```

MERGE: C CODE

```
merge(a,b,c,n)
int a[],b[],c[],n;
{
      int i,j,k;
      i=0;j=0;
      for(k=0;k<2*n;k++)
      {
            if (i>=n || j<n && a[i]>b[j]) {c[k]=b[j++];}
            else {c[k]=a[i++];}
      }
}
```

Three address code for inner loop

```
LOOP
   (LT CK LIMK CC1)
   (IF (NOT CC1) (GOTO EXIT))
   (LT AI LIMI CC2)
   (IF (NOT CC2) (GOTO L1))
   (LT BJ LIMJ CC3)
   (IF (NOT CC3) (GOTO L2))
   (LOAD A AI T2)
   (LOAD B BJ T3)
   (GT T2 T3 CC4)
   (IF (NOT CC4) (GOTO L2))
L1
   (LOAD B BJ T4)
   (STORE C T4 CK C)
   (ADD BJ 4 BJ)
   (GOTO L3)
L2
   (LOAD A AI T5)
   (STORE C T5 CK C)
   (ADD AI 4 AI)
L3
   (ADD CK 4 CK)
   (GOTO LOOP)
```

Percolation scheduling result

```
(1):
   ((LT CK LIMK CC1)
    (LT AI LIMI CC2)
    (LT BJ LIMJ CC3)
    (LOAD A AI T2)
```

```
    (LOAD B BJ T3)
    (GOTO (2)))
(2):
   ((GT T2 T3 CC4)
    (IF
     (NOT CC1)
     ((GOTO (E1)))
     ELSE
     ( (IF
        (NOT CC2)
        ((STORE C T3 CK C) (ADD BJ 4 BJ) (ADD CK 4 CK) (GOTO (1)))
        ELSE
        ( (IF
           (NOT CC3)
           ((STORE C T2 CK C) (ADD AI 4 AI) (ADD CK 4 CK) (GOTO (1)))
           ELSE
           ((GOTO (3)))) )) )))
(3):
   ((ADD CK 4 CK)
    (IF
     (NOT CC4)
     ((STORE C T2 CK C) (ADD AI 4 AI) (GOTO (1)))
     ELSE
     ((STORE C T3 CK C) (ADD BJ 4 BJ) (GOTO (1)))))
(E1):
```