Exploiting Ultra-fine Grain Parallelism for Machines With Parallel Pipelined Datapaths

Jie Gong
Daniel D. Gajski

Technical Report #92-112
December 20, 1992

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-8059

Abstract

This report examines ultra-fine grain machine parallelism determined by various hardware styles and constraints. Two major components are incorporated in our system: (1) A generalized parameterized architecture model which characterizes different design styles and constraints based on parallel pipelined machines. (2) A retargetable compiler which maps instruction parallelism to ultra-fine grain machine parallelism for target architectures. Basically the generalized parameterized model is used to specify different target machines, and the retargetable compiler compiles and schedules applications, codes written in high-level language, into control codes for given target machines. The resulting control codes are run through a simulator, after which dynamic statistics of the execution are recorded and the ultra-fine grain parallelism of target machines is assessed. A set of studies has been conducted to demonstrate how ultra-fine grain machine parallelism is affected by various hardware parameters and how performance is affected by both instruction parallelism and machine parallelism.

Contents

List of Figures

1 Introduction

Many researches have been working on exploiting instruction parallelism [NF84][WS84] [AKT86] [SV87][PS88]. The instruction parallelism of a program is a measure of the average number of instructions executable at the same time. The instruction parallelism is constrainted by the program i.e. by the number of data dependencies and the number of branches in relation to other instructions. Basically there are two ways of exploiting instruction parallelism. One is the static approach in which the selection of which instructions to issue in a given cycle is performed at compiler time. Another is the dynamic approach in which the instruction issuing is determined at run time. In static approach, since the instructions are issued during compiler time, aggressive optimization techniques such as loop unrolling, software pipelining[La88], percolation scheduling[Ni85] can be applied to exploit massive interbasic-block instruction parallelism. But in dynamic approach, instruction decode unit must look ahead at a sequence of instructions and issue each instruction based on instructions already issued and data dependencies between results and operands of instructions. It is very difficult, if not impossible, to encorporate those static parallelizing techniques into the hardware decode unit because of the complexity of these techniques and timing constraints of the hardware decode unit. Therefore, the static approach is more appropriate for exploiting global instruction parallelism.

To improve performance, not only instruction parallelism but also machine parallelism should be required. Machine parallelism is a measure of the ability of the processor to take advantage of the instruction parallelism. It is determined mostly by the processor organization i.e. by the number of actions or instructions that can be taken or executed at the same time. Performance is essentially affected by both instruction parallelism and machine parallelism. If the instruction parallelism in the applications exceeds the parallelism of the machine, then usually most of parallelism of the machine will be utilized and the performance will be limited by the parallelism of the machine. Similarly, if the instruction parallelism is less than the machine parallelism, then the performance will be limited by the instruction parallelism inherent in the applications[Jo89].

Machine parallelism is determined by processor organizations. Generally there are two datapath designs to achieve machine parallelism: (1) using parallel datapath, such as one containing multiple functional units; (2) using pipelined datapath, such as one containing pipelined functional units. In the first approach, several instructions may be issued per cycle and spatial parallelism is achieved. In the second approach, only one instruction may be issued per cycle, but the cycle time is shorter than the propagation time of the functional unit, thus temporal parallelism is achieved. Since these two approaches are orthogonal, we could have parallel pipelined datapath organizations. Of course the machine parallelism is not only affected by number and style of functional units, but it is also affected by other constituents of the processor such as size of register file, number of register file ports, number of memory ports, number of buses and connection style among different components.

In previous work, machine parallelism is abstracted by instruction set. For example, in

VLIW, *very long instruction word* machines, machine parallelism is determined by the predefined instruction sets. VLIW machines exploiting different amounts of machine parallelism would require different instruction sets because VLIW's that able to exploit more parallelism would require larger instructions. Basically VLIW machines exploits the machine parallelism at the register transfer level. This kind of parallelism is called *fine-grain* parallelism. The term "fine" is used to distinguish this level of parallelism from coarse-grain parallelism which refers to higher level of constructs such as loop iterations, parallel processors etc. Generally, the finer the granularity, the better the parallelism extraction. This is because irregular forms of parallelism are often not visible at coarser levels. Register transfer level is not the lowest level in which computation is carried out. It does not consider detailed architectural components such as internal latches, buses, register file ports which implement the actual data transfers. Predefined instruction set may not fully expose the parallelism available in machine organizations. To exploit *ultra-fine grain* machine parallelism existing in the level below instruction set, instead of generating parallel instructions, we produce control codes which are used for issuing parallel micro-actions for datapath per cycle. Because of more flexibilities existing in the control code than that in the instruction set, more machine parallelism can be exposed at ultra-fine grain level. Furthermore, effect of various processor constituents on the machine parallelism can be studied at this level whereas the effect of many processor components such as register file ports, buses etc. is ignored at instruction set level since the instruction set hardly reflects those organization aspects.

To exploit ultra-fine grain machine parallelism offered by the machine organization, we must consider the detailed target machine organizations which contain functional units, storages elements and interconnection units. In order to characterize various machine styles and constraints, a generalized parameterized architecture model is proposed. The model is based on RISC-like load/store architecture extended with a parallel pipelined datapath. It can have multiple functional units, a multi-port register file or memory, and multiple buses with arbitrary bus connections among different components. Each functional unit can be of single or multiple cycle latency and can be non-pipelined or pipelined. A concrete architecture can be instantiated from this generic model by specifying values for various parameters.

In our approach, machine parallelism is constrainted by the target architecture specified by the generalized model and instruction parallelism and ultra-fine grain parallelism is exploited by a retargetable compiler. The compiler has two major phases which exploit instruction parallelism and ultra-fine grain parallelism respectively. In the first phase of optimization, percolation scheduling, loop pipelining and memory disambiguation are applied to the applications to achieve massive instruction parallelism across basic blocks. In the second phase each pseudo-basic block obtained from the previous phase is scheduled by a generalized mapping algorithm onto the target machine and corresponding control codes for the target machine are generated.

The remainder of this report is organized as follows. In next section we describe the generalized architectural model. In section 3 the retargetable parallel compiler is shown and algorithms for mapping instruction parallelism down to machine parallelism is presented. In

section 4, an overview of our system is shown and a set of experimental results are discussed Finally a conclusion is drawn in section 5.

2 The Architecture Model

The generic parameterized architecture model is described as follows:

- There are n functional units, n_i functional units of type T_i, $i = 1, \ldots, k, n = \sum_{i=1}^{k} n_i$.
- Each functional unit of type T_i has latency of N_i clock cycles, $i = 1, \ldots, k$. Latency is defined as the delay from input ports to the output port of functional unit.
- Each functional unit may or may not have two input latches and one output latch. Each functional unit can be pipelined or non-pipelined. Each functional unit may have bypass route around its output latch.
- There is one register file, with P ports and r registers. Reading from and writing to the register file are assumed to take one clock cycle.
- There are one memory module, with m ports. Each port has one MAR and one MBR latch associated with it, and memory module has latency of M clock cycles.
- Units can be interconnected with one or more buses. Interconnections are specified by a connection table M. The row indices of the matrix are buses. The column indices of the matrix are components. Entry $M[i, j]$ has value '1' if there is connection between bus_i and $component_j$, otherwise $M[i, j]$ has value '0'.

This proposed architecture model is general since it incorporates various design constraints and design styles. Based on a bus_oriented topology, the architecture displays some regularities. It is also very flexible since architecture components can be added or removed simply by redefining values for hardware parameters.

Fig. 1 shows a concrete architecture which consists of three functional units: two ALUs and one multiplier. ALU1 is a single cycle unit with bypass route around its output latch. ALU2 and MUL3 are multi-cycle functional units which can be pipelined or non-pipelined. L1–L3 are left operand latches in front of functional units while R1–R3 and O1–O3 are right operand latches and output latches respectively. OUT1 denotes the output port of ALU1. There is one register file which has two ports P1 and P2 and a memory which has one port associated with two latches: MAR and MBR. The memory has multiple cycle latency. The interconnection table for this concrete architecture is shown in Fig. 2.

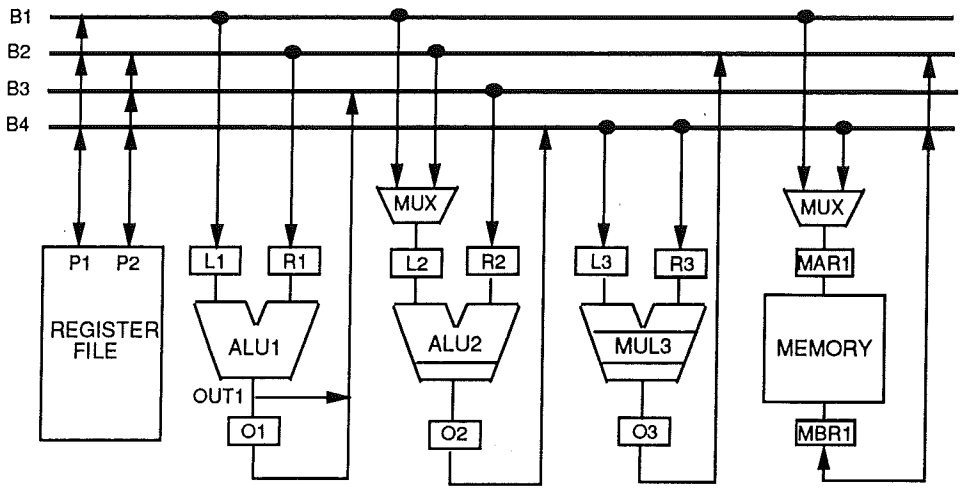

Figure 1: An architecture instance

	L1	R1	O1	Out1	L2	R2	O2	L3	R3	O3	P1	P2	MAR1	MBR1
Bus1	1				1						1		1	
Bus2		1			1					1	1	1		1
Bus3			1	1		1						1		
Bus4							1	1	1		1	1	1	1

"1s" indicate connections

Figure 2: Connection table for architecture in Fig. 1

3 The Parallel Compiler

The retargetable parallel compiler maps applications onto given target architectures. It is composed of three parts: (1) Gnu C front end compiler; (2) VLIW (Very Long Instruction Word) compiler; (3) UP (Ultra-fine-grain Parallel) compiler.

3.1 Gnu C Compiler

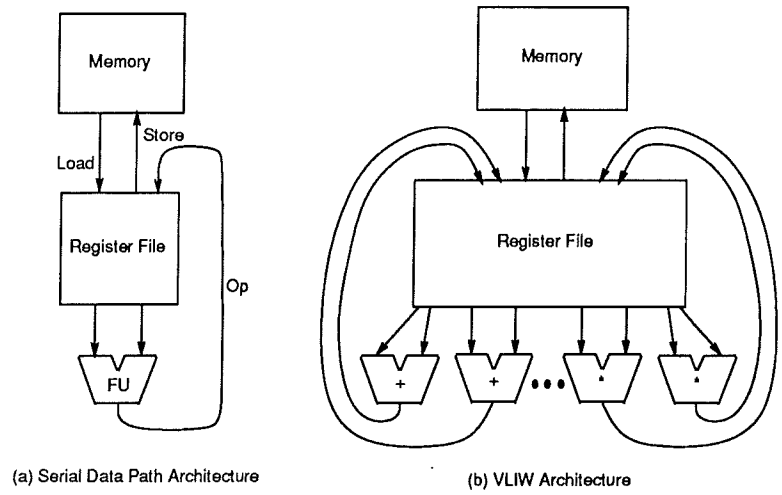

Figure 3: Two abstract architectures

The Gnu C Front End Compiler compiles an application written in the C language into a register transfer level intermediate format which is serial three-address code (Each instruction only consists of one three-address operation). The three-address operation (two source operands and one result) is based on the operations of load/store architecture in which only loads and stores can access memory and other operations solely work on the registers. The three-address operation has following formats:

- des ← src1 op src2; ("arithmetic or logic" operation)
- mem(addr) ← src; ("store" operation)
- des ← mem(addr); ("load" operation)
- **if** cond **goto** label; ("conditional jump" operation)
- **goto** label; ("unconditional jump" operation)

"des", "src", and "addr" are either registers or constants. "cond" is a one-bit register. "label" is the label of each three-address operation.

In this stage register allocation has been done and the compiler tries to use the available registers as much as possible to reduce load/store operations. The serial three-address code produced by the Gnu C Compiler can be viewed as the instructions executed on the architecture shown in Fig. 3 (a).

3.2 VLIW Compiler

The VLIW Compiler takes serial three-address code as input and generates parallel three-address code in which one instruction can consist of multiple three-address operations. Advanced compiler techniques (percolation scheduling, loop pipelining, memory disambiguation, etc.) are implemented in the VLIW compiler to achieve the fine grain parallelism across basic blocks[Po91]. In order to exploit massive instruction parallelism in the code, an unlimited number of resources is assumed and each operation is assumed to take one clock cycle. The architecture on which the parallel three-address code is executed is shown in Fig. 3 (b). There are multiple functional units and one register file with multiple ports. The register file and functional units are fully connected, i.e. every register file port has connections to inputs/output of every functional unit.

3.3 Ultra-fine-grain Parallel Compiler

The ultra-fine-grain parallel (UP) compiler takes either serial three-address code or parallel three-address code as input and maps it onto the target architectures specified by the architecture model described in section 2. The UP compiler compiles a register-transfer level description down to microcode which essentially describes movement between storage elements, e.g. movement from register to latch, from latch to latch, from latch to memory location etc. Those microcode serves as the control code for the given target architecture.

The UP parallel compiler consists of three main parts including a basic block recognizer, a dependence graph builder and a scheduler and binder shown in the dash rectangles of Fig. 4. The dependence graph builder and the scheduler and binder consider different architectural parameters in their algorithms. The algorithms used by each part will be described in following sections.

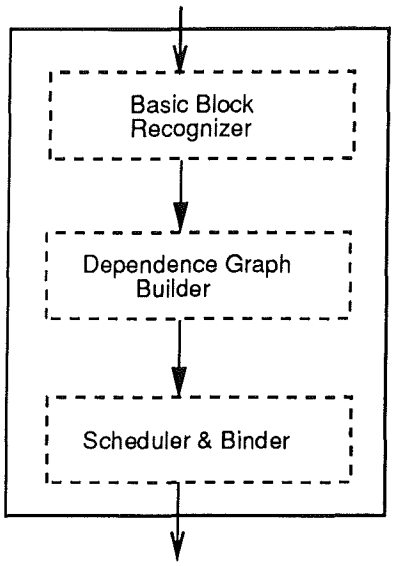

Figure 4: The ultra-fine-grain parallel compiler

3.3.1 Algorithm for Basic Block Recognizer

The basic block recognizer takes serialized program graph as input and enumerates basic blocks of the program. A program graph is a directed graph $G = (V, E)$. For serial three-address code, each node in V consists of only one three-address operation. This type of node is called a serial node. For parallel three-address code, each node in V may contain multiple three-address operations which can be executed concurrently. This type of node is called a parallel node. Each edge in E represents execution sequencing. If there is an edge from node a to node b, it means that node a is a predecessor of node b (or node b is a successor of node a) and node a needs to be executed before node b. A parallel node in G can be split into multiple serial nodes which have the same semantics as the parallel node. This is called a serialization process. A program graph G consisting of only serial nodes is called a serialized program graph.

In the following algorithm, procedure $NUM\_OF\_PRED(m)$ returns the number of predecessors of m; $NUM\_OF\_SUCC(m)$ returns the number of successors of m; $IS\_STARTING\_NODE(m)$ returns TRUE if m is the starting node of the program graph; $SUCC(m)$ returns the unique successor of node m when m has only one successor; $NEW\_LIST(m)$ creates a new list with m as the first element.

Basically what the algorithm does is to find the entry node of a basic block first, then list the nodes which belong to the same basic block until entry node of a new basic block is found.

Algorithm: Find basic blocks of a serialized program graph G

```
for each node n ∈ V of G do
    m = n;
    if (NUM_OF_PRED(m) > 1) or (IS_STARTING_NODE(m)) then
        /* create a new basic block with m as the entry node */
        b = NEW_LIST(m);
        while (NUM_OF_SUCC(m) = 1) and (NUM_OF_PRED(SUCC(m)) = 1) do
            append SUCC(m) to b;
            m = SUCC(m);
        endwhile
    endif
endfor
```

3.3.2 Algorithms for Dependence Graph Builder

The dependence graph builder takes each basic block as input and produces a data dependence graph (DDG) for the list of three-address operations of a basic block. The DDG is a directed acyclic graph $G = (V, E)$. Each node in V is a three-address operation. There are three types of dependence edges in E: (1) flow dependence (write-read) edges; (2) anti-dependence (read-write) edges; (3) output dependence (write-write) edges.

In the following algorithm, L is a list of three-address operations. $FIRST(L)$ returns the first element of L. $PREV(n)$ or $NEXT(n)$ returns the previous or next element of n in list L. The previous element of the first element is ϕ, as is the next element of the last element. $SRC(n)$ returns the two source variables of the three-address operation n. $DEST(n)$ returns the destination variable of n. $ADD\_NODE(DDG, n)$ adds a node containing n to DDG. $ADD\_EDGE(DDG, m, n, TYPE)$ builds a edge of type "$TYPE$" from node m to n if there is no any path from m to n in the DDG.

Basically the algorithm creates a node for each three-address operation in DDG. After that the algorithm builds the flow dependence edges, output dependence edges and anti-dependence edges among the nodes in DDG by scanning each operation and its previous operations in the list L. An edge is built between two nodes only if there is no path between those two nodes in order to avoid redundant edges.

Algorithm: Build DDG for a list L of three-address operations

```
DDG = φ;
n = FIRST(L);
while n ≠ φ do
    /* build a node */
```

```
ADD_NODE(DDG, n);
/* build flow dependence edge */
m = PREV(n);
while m ≠ φ do
    if (SRC(n) ∩ DEST(m)) ≠ φ then
        ADD_EDGE(DDG, m, n, flow_dependence);
        exit while loop;
    else
        m = PREV(m);
    endif
endwhile
/* build output dependence edge */
m = PREV(n);
while m ≠ φ do
    if (DEST(n) ∩ DEST(m)) ≠ φ then
        ADD_EDGE(DDG, m, n, output_dependence);
        exit while loop;
    else
        m = PREV(m);
    endif
endwhile
/* build anti-dependence edge */
if m = φ then m = FIRST(L) endif
k = PREV(n);
while k ≠ PREV(m) do
    if (DEST(n) ∩ SRC(k)) ≠ φ then
        ADD_EDGE(DDG, k, n, anti_dependence);
    else
        k = PREV(k);
    endif
endwhile
/* do for next node */
n = NEXT(n);
endwhile
```

3.3.3 Algorithms for the Scheduler and Binder

The scheduler and Binder takes a DDG as input and does scheduling and binding for the DDG based on the given parameterized target architectures.

Each node in DDG is assigned a weight which is the number of executing stages of the operation in the node. The executing stages of a operation depends on the type and latency of the unit used. It also depends on whether there are input/output latches for the unit.

For example, assume some functional unit has a clock cycle latency of n, Suppose operation 'op' is executed on this functional unit. Then the weight of three-address operation "R1 $\leftarrow$ R2 op R3" will be n if there are no input/output latches for the functional unit, $n+1$ if there are only input latches, $n+2$ if there are both input and output latches. For a memory with latency of n clock cycles, the executing stages of the LOAD operation will be $n+2$, the executing stages of the STORE operation will be $n+1$. Basically based on the parameterized target architecture, the weight of each operation can be formulated. After this weight assignment process, the DDG becomes a weighted DDG $G_{weighted} = (V, E, W)$, $|W| = |V|$.

The ASAP(As Soon As Possible) value, ALAP (As Late As Possible) value and Mobility for each node in the weighted DDG is computed since they are used as priority functions in the scheduling and binding algorithm. The following algorithm assigns each source node an ASAP value, each sink node an ALAP value, then traverses the weighted DDG to get ASAP and ALAP values for each node. The mobility for each node can be easily calculated using ASAP and ALAP values.

In the following algorithm, $PRED(v)$ or $SUCC(v)$ returns a set of all immediate predecessors or successors of v. $NO\_ASAP(v)$ or $NO\_ALAP(v)$ returns $TRUE$ if v has not be assigned an ASAP value or an ALAP value. $ALL\_PRED\_HAVE\_ASAP(v)$ or $ALL\_SUCC\_HAVE\_ALAP(v)$ returns $TRUE$ if all predecessors or successors of v have their ASAP or ALAP values calculated respectively. $MAX\_ASAP(PRED(v_i))$ returns the maximum ASAP value among the set of predecessors of node v_i. $MIN\_ALAP(SUCC(v_i))$ returns the minimum ALAP value among the set of successors of node v_i. $ABS(x)$ returns the absolute value of x.

Algorithm: Compute ASAP, ALAP, and Mobility for a weighted DDG

```
/* compute As Soon As Possible values */
for each node v_i ∈ V do
    if PRED(v_i) = φ then
        ASAP(v_i) = 1;
    endif
endfor
for each node v_i ∈ V do
    if (NO_ASAP(v_i)) and (ALL_PRED_HAVE_ASAP(v_i)) then
        ASAP(v_i) = MAX_ASAP(PRED(v_i)) + Weight(v_i);
    endif
endfor
/* compute As Late As Possible values */
for each node v_i ∈ V do
    if SUCC(v_i) = φ then
        ALAP(v_i) = length_of_critical_path − Weight(v_i);
    endif
```

endfor
for each node $v_i \in V$ **do**
  if (NO_ALAP(v_i)) and (ALL_SUCC_HAVE_ALAP(v_i)) **then**
    $ALAP(v_i) = MIN\_ALAP(PRED(v_i)) - Weight(v_i)$;
  endif
endfor
/* compute mobilities */
for each node $v_i \in V$ **do**
  $Mobility(v_i) = ABS(ALAP(v_i) - ASAP(v_i))$;
endfor

Basically the algorithm for the scheduler and binding is a variation of list scheduling. It incorporates various architecture parameters into scheduling and binding process.

Assume that the target architecture has n resources (including functional units and memory), n_i resources of type T_i. i = 1, ..., k. n = $\sum_{i=1}^{k} n_i$. The algorithm uses a priority list PList for each type resource of T_i, i = 1, ..., k. These lists are denoted by the variables $PList_{T_1}, \ldots, PList_{T_k}$. There is also a priority list $BusPList$ used to queue the operations competing for the bus resources. The $BusPList$ is a list of tuples $< op, resrc >$ which means that operation op is allocated to the resource $resrc$. Initially all priority lists are empty.

The priority lists are maintained as follows: first, nodes are sorted in ascending order of their mobilities. All nodes with identical mobilities are sorted in increasing order of their ALAP values. At last the unique sequence number of each operation is used to break tie if there is one.

There is one queue for each type of resource. These queues are denoted by the variables $RQueue_{T_1}, \ldots, RQueue_{T_k}$. Initially there are n_i resources of type T_i in $RQueue_{T_i}$. For example, suppose three ALUs exist in the given architecture, then initially there are three resources $ALU1, ALU2, ALU3$ in $RQueue_{ALU}$.

There are two auxiliary queues: $RRQu$ (Resource Ready Queue) and $EDQu$ (Edge Deletion Queue). $RRQu$ is used to queue "at which control step the occupied resources can be released". It is a queue of tuples $< resource, Cstep >$. $EDQu$ is used to queue "at which control step the edges of DDG can be deleted from the DDG". It is a queue of tuples $< edge, Cstep >$. Initially both $RRQu$ and $EDQu$ are empty.

A ready node in DDG is one whose in-degree is zero i.e. it has no incoming edges. $INSERT\_READY\_OPS(DDG, PList_{T_1}, \ldots, PList_{T_k}, Cstep)$ scans the set of nodes of DDG, determines if any of the nodes are ready at control step $Cstep$, deletes the ready node from DDG (but **not** its associated outcoming edges), and appends the ready nodes to one of the priority lists based on their operation types.

$INSERT\_OP(op, r, BusPList)$ inserts the tuple $< op, r >$ into $BusPList$. Here opera-

tion *op* has been bound to resource r. $INSERT\_READY\_RESOURCES(RRQu, RQueue_{T_1}, \ldots, RQueue_{T_k}, Cstep)$ scans queue RRQu, finds out all resources released at the control step $Cstep$, appends them to one of the resource queues based on their resource types. $DELETE\_EDGE(EDQu, DDG, Cstep)$ scans queue $EDQu$, finds out all edges need to be deleted from DDG at control step $Cstep$, then deletes them from DDG.

$DELETE(L, x)$ returns the new L in which x has been removed out. $INSERT(L, x)$ returns the new L in which x has been inserted in. $FIRST(L)$ returns the first element of L. $NEXT(x)$ returns the next element of x in L.

The connection of the architecture is represented by a matrix M. The row indices of the matrix are buses. The column indices of the matrix are components. Entry $M[i, j]$ has value '1' if there is connection between bus_i and $component_j$, otherwise $M[i, j]$ has value '0'. There is another matrix which is called bus reserve table (BRT). The row indices of BRT are buses. The column indices of BRT are control steps. $BRT[i, j]$ is '1' if bus_i is reserved at control step j, otherwise it is '0'.

$BUS\_RESERVED(op, r, Cstep, BRT)$ returns $TRUE$ and sets '1' to the reserved entries of BRT if the bus resources required by *op* have be fulfilled in the BRT, otherwise returns $FALSE$. The bus reservation process uses a lookahead approach starting from current step $Cstep$. For example, suppose resource r is a functional unit and has both input and output latches. Basically the reservation procedure will try to look up column $Cstep$ and column $Cstep + latency + 1$ of BRT as well as connection matrix M to decide if there are buses available for scheduling the operation *op* since *op* will consume bus resources at control step $Cstep$ and control step $Cstep + latency + 1$ to move data between register file and latches.

$SCHEDULE\_OP(op, r, Cstep)$ binds operation *op* to resource r and reserved bus resources and creates the scheduling sequence for *op* starting from control step $Cstep$. For example, suppose r has both input and output latches and has latency of 1 clock cycle and the required bus resources are satisfied, the scheduling sequence for *op* will be (1) In step $Cstep$ contents of source registers are moved to input latches of r. (2) In step $Cstep + 1$, 'L_r op R_r' are moved to O_r where L_r, R_r and O_r are left, right, output latches of r respectively. (3) In step $Cstep + 2$, content of output latch is moved to the destination register.

$RESOURCE\_AVAILABILITY(r, Cstep)$ returns the control step at which the occupied resource r is released based on the resource type, latency, and style(e.g. pipelined/non-pipelined) as well as the current control step $Cstep$. For example, suppose r is a functional unit with latency of n, r will be available at one step later of the control step where r is used if r is pipelined. If r is not pipelined, r will be available n steps later of the step where r is used.

$EDGE\_DELETION(op, Cstep)$ returns the control steps at which outcoming edges of *op* in DDG should be deleted based on the allocated resource type, latency, style, current control step $Cstep$ as well as types of the edges. For example, a flow dependence edge of

op can be deleted only after the step where the destination register of *op* is written while a anti-dependence edge can be deleted after the source registers of *op* are read.

$x, < y_1, \ldots, y_e >, < op, r >, < op_{next}, r_{next} >, i, Cstep, E_1, \ldots, E_e$ are temporal variables.

Algorithm: Scheduling and binding of DDG for Target Architecture

```
Cstep = 0;
INSERT_READY_OPS(DDG, PList_T1, ..., PList_Tk, Cstep+1);
while (PList_T1 ≠ φ)or...or(PList_Tk ≠ φ)or(BusPList ≠ φ)or(DDG ≠ φ) do
    Cstep = Cstep + 1;
    /* assign resources to ready lists */
    for i = 1 to k do
        while RQueue_Ti ≠ φ do
            r = FIRST(RQueue_Ti);
            if PList_Ti ≠ φ then
                op = FIRST(PList_Ti);
                RQueue_Ti = DELETE(RQueue_Ti, r);
                PList_Ti = DELETE(PList_Ti, op);
                INSERT_OP(op, r, BusPList);
            else
                break;
            endif
        endwhile
    endfor
    /* assign buses to bus queue */
    < op, r > = FIRST(BusPList);
    while < op, r > ≠ φ do
        < op_next, r_next > = NEXT(< op, r >);
        if BUS_RESERVED(op, r, Cstep, BRT) then
            SCHEDULE_OP(op, r, Cstep);
            x = RESOURCE_AVAILABILITY(r, Cstep);
            RRQu = INSERT(< r, x >, RRQu);
            /* assume 'op' has e outcoming edges E_1, ..., E_e */
            < y_1, ..., y_e > = EDGE_DELETION(op, Cstep);
            for t = 1 to e do
                EDQu = INSERT(< E_t, y_t >, EDQu);
            endfor
            BusPList = DELETE(BusPList, < op, r >);
        endif
        < op, r > = < op_next, r_next >
    endwhile
    /* prepare for next control step */
    INSERT_READY_RESOURCES(RRQu, RQueue_T1, ..., RQueue_Tk, Cstep+1);
```

DELETE_EDGE(EDQu, DDG, Cstep+1);
INSERT_READY_OPS(DDG, $PList_{T_1}, \ldots, PList_{T_k}$, Cstep+1)
endwhile

4 System Overview and Experimental Results

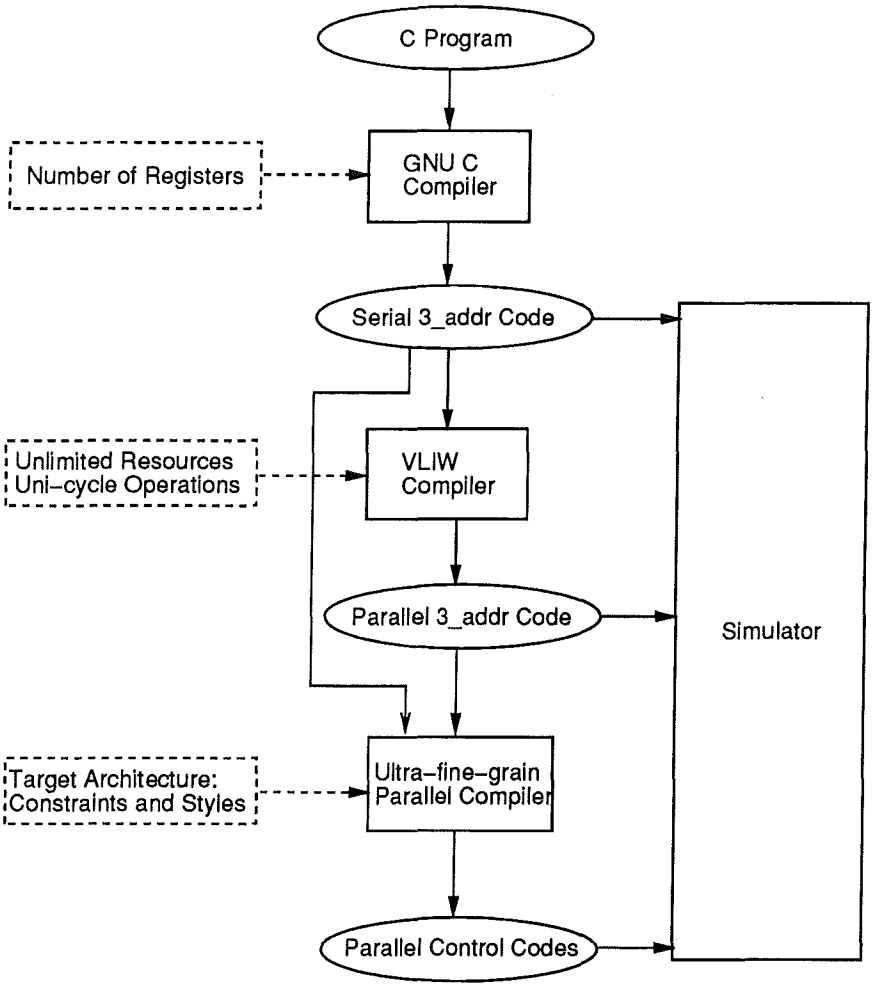

Figure 5: The overall system

Fig. 5 shows a block diagram of our system. The design constraints and styles specified in the architecture model will be fed into different stages of the parallel compiler. A bypass is provided so that application can be compiled onto the parameterized target architecture with or without going through the VLIW compiler. Therefore code with either less instruction parallelism or more instruction parallelism can be mapped onto the given target architecture. A simulator is implemented to mimic the execution of the serial three-address code, parallel

three-address code as well as the parallel microcode. Based on the input values of program variables specified in the memory locations, the simulator will run through the codes and produce the resulting values in memory locations. Also, the simulator records dynamic statistics such as the number of cycles executed, and the number of registers used. The absolute performance is measured by the time used to execute the control code, which is the number of clock cycles executed times clock cycle duration. For relative performance measuring, speed_up is used. Speed_up is defined as $x \div y$ where x is the execution time on the reference target machine, and y is the execution time on the target machine whose performance is to be rated.

To see how various architecture constraints affect the machine parallelism, several orthogonal experiments have been conducted. Four applications are chosen: the fast Fourier transformation FFT, a cosine computation COS from Stanford benchmarks, and two Livermore loops LL1 and LL19.

4.1 Bus Constraints

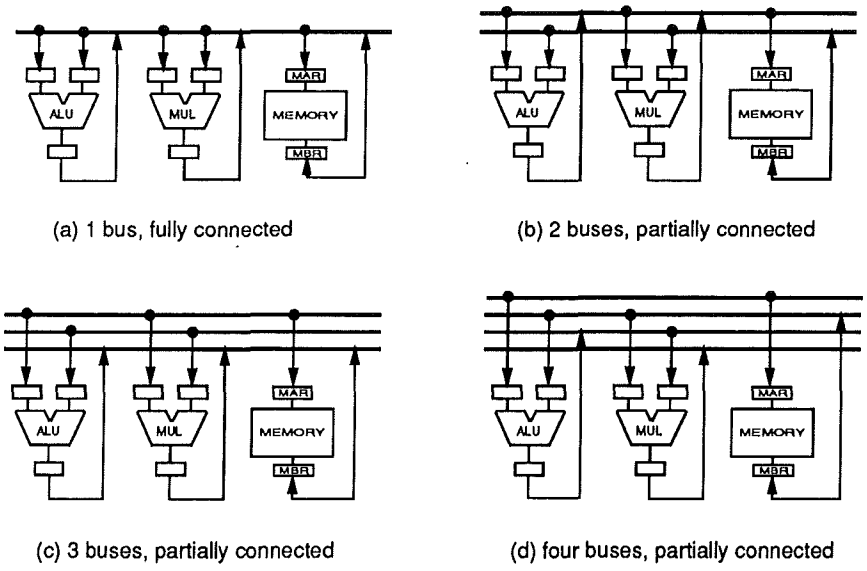

Figure 6: Several bus configurations

After giving a concrete architecture, we study how bus constraints affect the machine parallelism. The given architecture has one ALU and one Multiplier. Both are pipelined. The ALU has a 1 clock cycle latency while the multiplier has a 3 cycle latency. Both units have input and output latches. Therefore the ALU operation has 3 pipeline stages (1 stage from register file to input latches, 1 stage from input latches to output latch, 1 stage from output latch back to register file), the multiplication has 5 pipeline stages (1 stage from

register file to input latches, 3 stages from input latches to output latch, 1 stage from output latch back to register file). The architecture has one memory with a latency of 2 cycles. The memory has one port associated with MAR and MBR latches. Therefore the LOAD operation takes 4 cycles (1 cycle from register file to MAR, 2 cycles to load data into MBR, 1 cycle from MBR back to register file). The STORE operation takes 3 cycles (1 cycle from register file to MAR and MBR, 2 cycles to write data into memory). There is a register file which has same number of ports as buses. Register file ports are fully connected to the buses. Functional units and memory can be fully or partially connected to the buses. Fig. 6 shows several bus configurations of those we examined in the experiment.

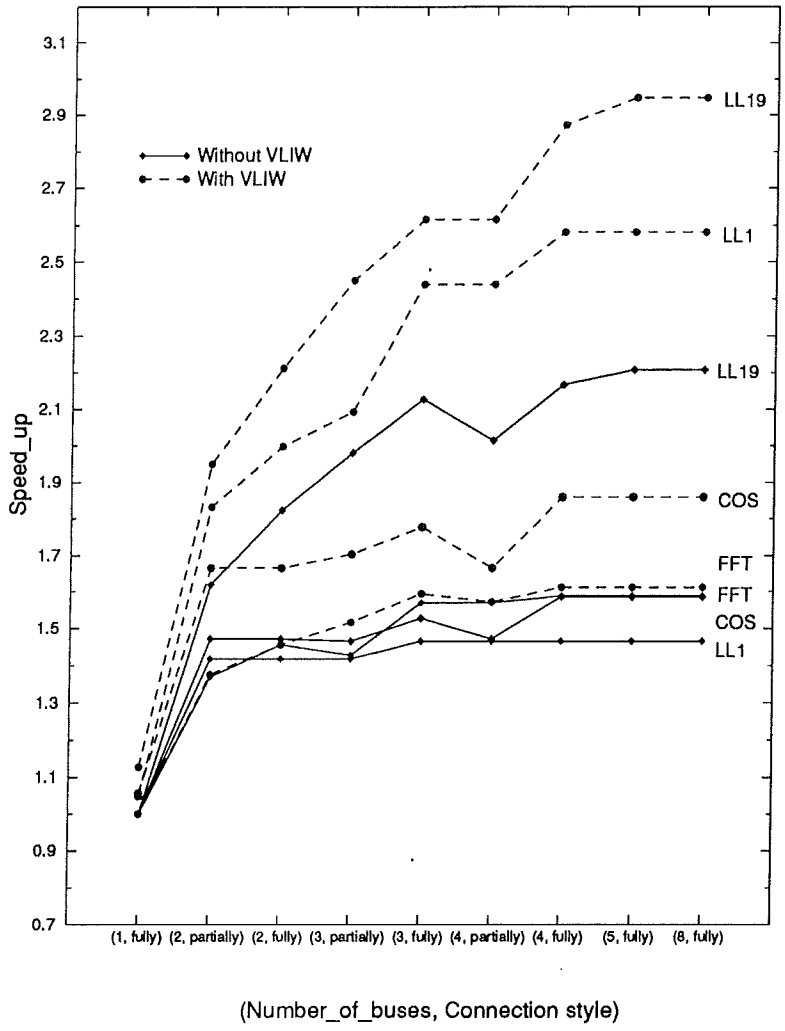

Figure 7: A study for bus constraints

Fig. 7 shows how machine parallelism is affected by bus constraints. From the graph we can observe that for such an architecture, one or two buses will limit the machine parallelism. On the other hand more than four buses in the architecture will not help improving the

machine parallelism any more due to the number of memory ports used. Basically one memory port in this given architecture creates a bottleneck after the number of buses are more than four. Therefore machine parallelism does not improve even when number of buses are increased afterwards. This graph also shows that not only machine parallelism but also instruction parallelism affects the performance. With same machine parallelism, more instruction parallelism will lead to better performance.

4.2 Functional Unit Constraints

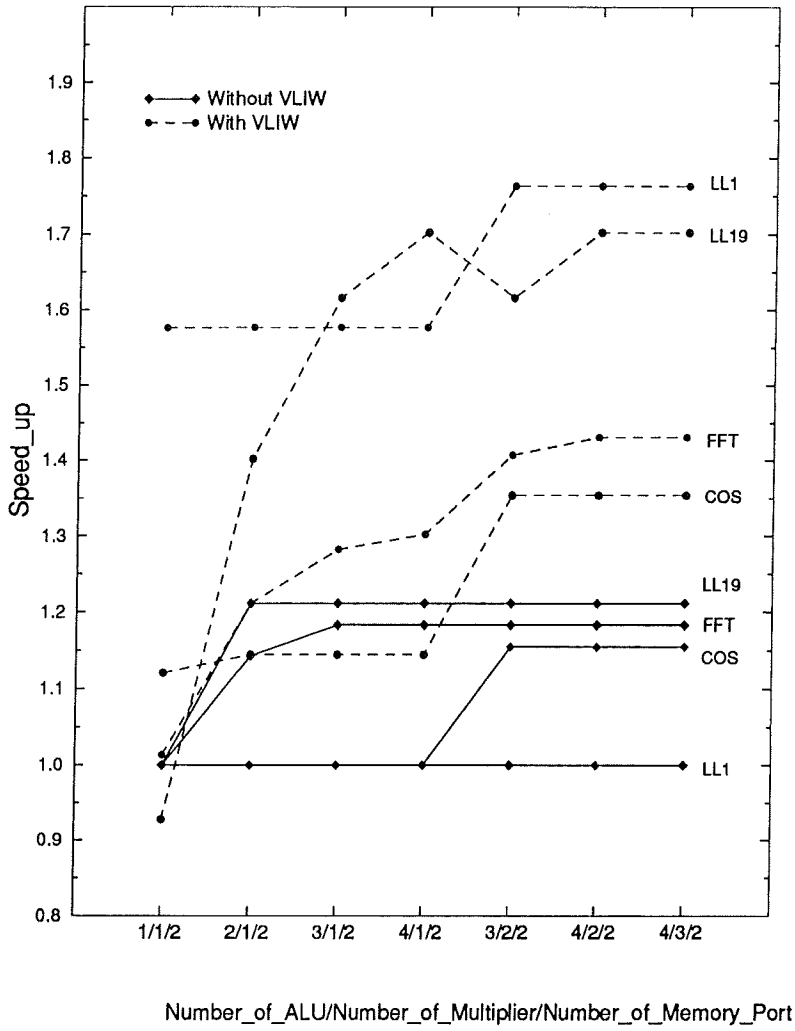

Figure 8: A study for functional unit constraints

To study how the number of functional units affect the machine parallelism. we consider an architecture which has ALUs with 1 cycle latency, multipliers with 3 cycle latency, one memory with a 2 cycle latency and 2 ports. Every unit has input-output latches. The ALUs

and multipliers are not pipelined. There are 4 buses which are fully connected with units and ports. There is a register file with 4 ports.

Fig. 8 shows the results obtained from the simulation. For some applications the performance improves along with the increasing in number of ALUs. For other applications increasing the number of multipliers improves the performance. This shows that for different types of applications, different types of machine parallelism should be provided in order to improve performance.

4.3 Memory Port Constraints

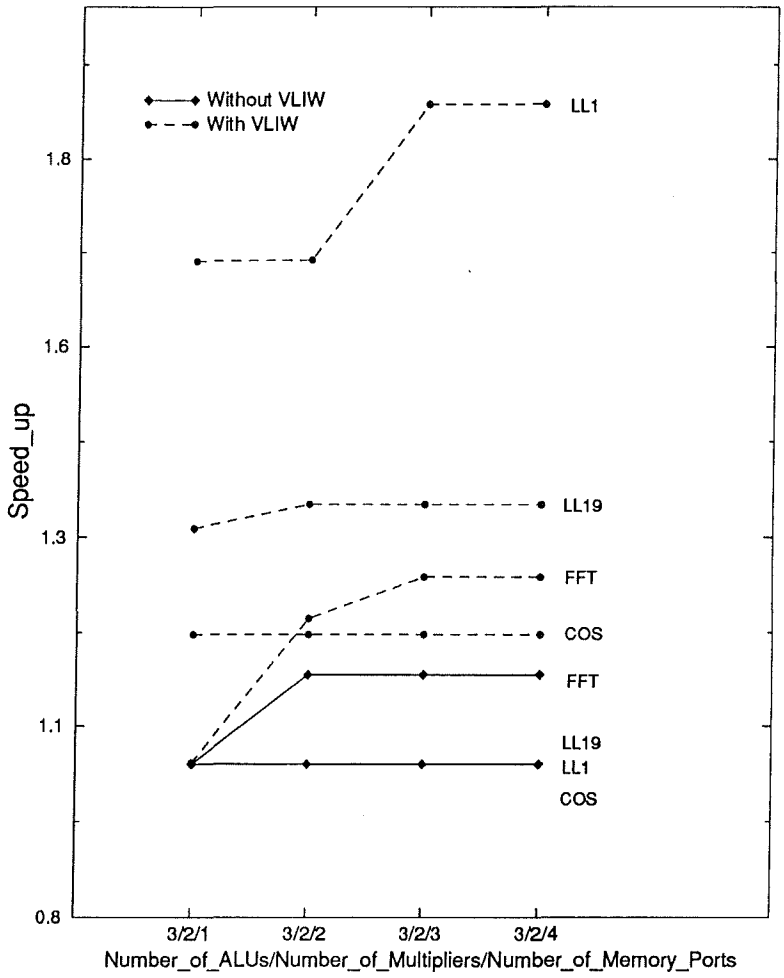

Figure 9: A study for memory port constraints

In this experiment, we study how number of memory ports affects the design performance. The architecture we use has 2 multipliers with 3 cycle latency, 3 ALUs with 1 cycle latency.

1 memory with 2 cycle latency. Every unit has input-output latches. ALUs and multipliers are not pipelined. There are 3 buses that are fully connected with units and ports. There is a register file with 3 ports.

Fig. 9 shows the results obtained from the simulation. We notice that increasing the number of memory ports helps applications which have more memory accesses such as the FFT and LL1. For applications with few memory accesses such as LL19 and COS, increasing the number of memory ports does not help the performance improvement at all. Therefore when increasing different dimensional machine parallelism, application characteristics should be considered. Otherwise some machine parallelism may be wasted.

4.4 Temporal Parallelism vs. Spatial Parallelism

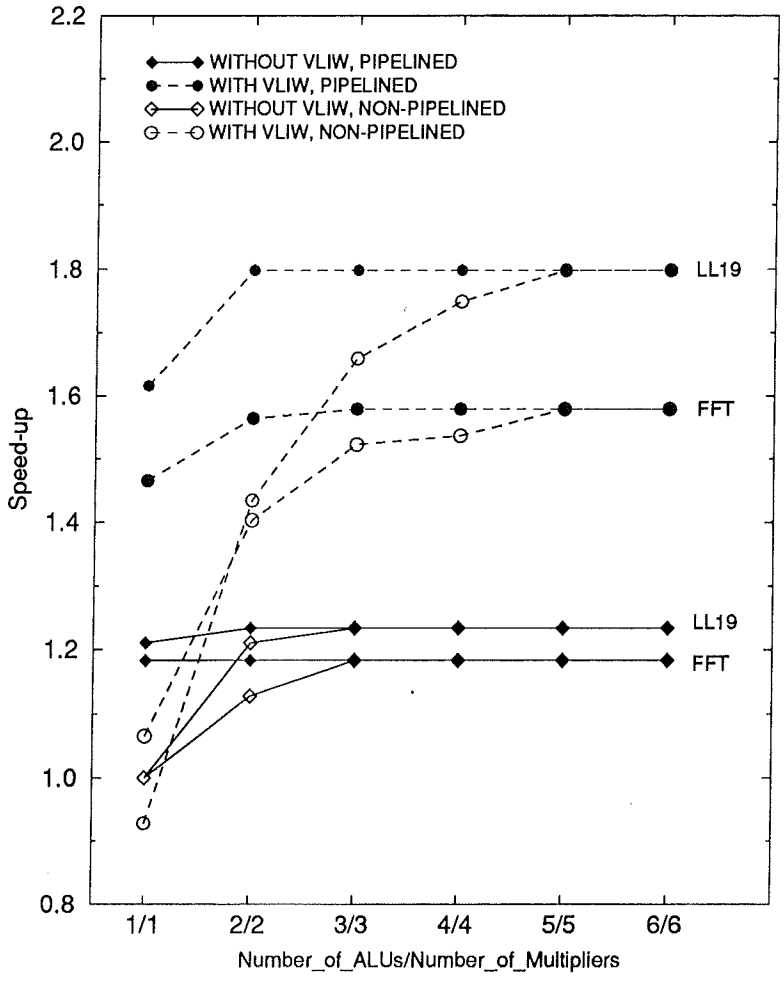

Figure 10: A study for pipeline vs. nonpipeline

There are two types of parallelism: temporal parallelism which is achieved by pipeline technique and spatial parallelism which is achieved by multiple functional units. Here we compare those two types of parallelism. The architecture we use has 6 buses, fully connected to units and ports. There is a memory with 4 ports, 1 register file with 6 ports. ALU has 1 cycle latency, multiplier has 3 cycle latency and memory has 2 cycle latency. Every unit has input-output latches.

Fig. 10 shows the results obtained from the simulation. From the graph we observe that a design of 1 pipelined ALU and 1 pipelined Multiplier results in machine parallelism similar to a design with 2 or 3 non-pipelined ALUs and 2 or 3 non-pipelined multipliers. Basically to achieve similar performance, pipelining uses less hardware.

4.5 Pipeline Constraints

We study how pipeline stages affect the machine parallelism. The architecture we use has 6 buses, fully connected to units and ports, 1 memory with 4 ports, 1 register file with 6 ports.

Fig. 11 shows the result from a non-pipelined (1 stage pipelined) situation to an 8 stage pipelined situation. When the number of stages increases, the clock cycle duration decreases. We assume the clock cycle duration is reduced in a factor of $(n+2) \div 2$ when $n \geq 2$ and n is the number of pipeline stages. The result shows that the performance improves along with the increment of the number of pipeline stages until some point where the performance starts to degrade. Basically after those saturated points, the loss of performance due to pipeline dependences introduced by the number of pipeline stages outweighs the gain obtained from more parallelism offered by the number of pipeline stages. For more functional units, the saturated points are reached faster due to more parallelism provided in the datapath.

5 Conclusion

We have studied ultra-fine grain machine parallelism available at the level below instruction set. Our system consists of a generalized architecture model which is used to specify the target machines and a retargetable compiler which does the mapping from fine grain instruction parallelism to ultra-fine grain machine parallelism. Through a simulator, we are able to evaluate the machine parallelism of the design. A set of experiments are conducted to show how machine parallelism is affected by different hardware parameters. The experimental results also show how performance is affected by both instruction parallelism and machine parallelism.

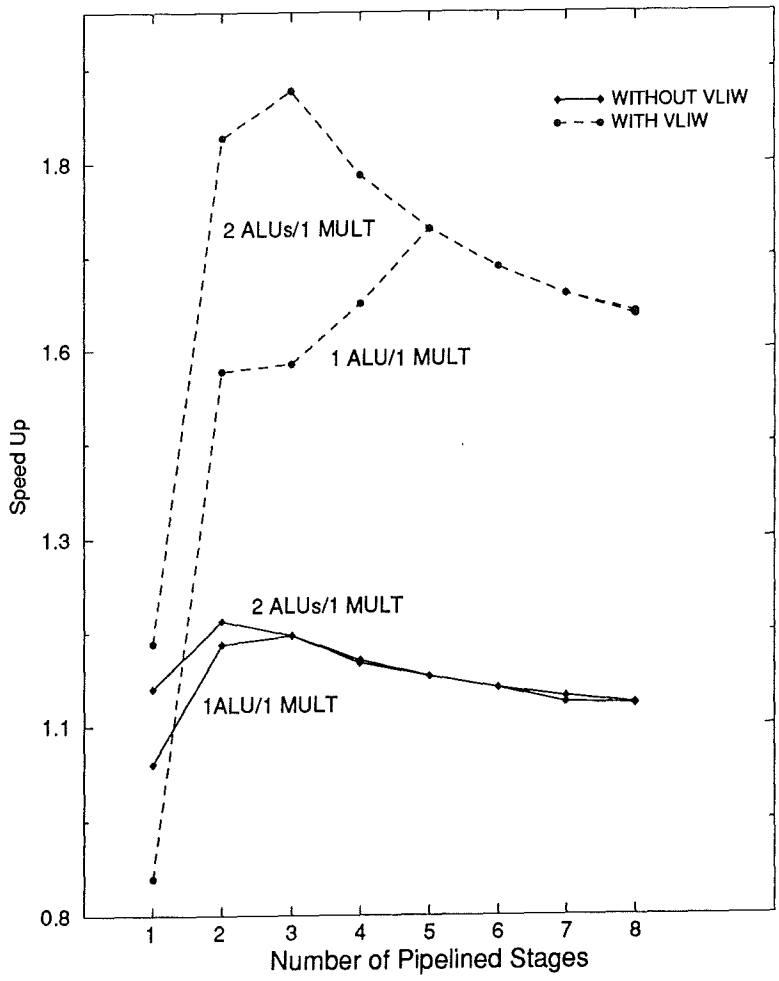

Figure 11: A study for pipeline constraints on LL19

6 Acknowledgements

Our system is built on the VLIW compiler implemented in Dept. of Information and Computer Science, UC Irvine by Roni Potasman et al. We would like to thank Dr. Roni Potasman and Prof. Alex Nicolau for their discussions and suggestions for this work. We are also grateful for the support from the Semiconductor Research Corporation (grant #92-DJ-146).

7 References

[AKT86] R. D. Acosta, J. Kjelstrup and H. C. Torng, "An instruction issuing approach to enhancing performance in multiple functional unit processors," IEEE Trans. Comput., vol. C-35, Sept. 1986.

[Br91] M. Breternitz Jr, "Architecture synthesis of high-performance application-specific processors," Ph.D thesis, Carnegie Mellon University, April 1991.

[JaMu91] R. Jain, A. Mujumdar, A. Sharma and H. Wang, "Empirical evaluation of some high-level synthesis scheduling heuristics," Proceedings of the 28th Design Automation Conference, 1991.

[Jo89] N. P. Jouppi, "The distribution of instruction-level and machine parallelism and its effect on performance," IEEE Trans on Comput, vol. 38, no. 12, Dec. 1989.

[La88] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," Proceedings of the SIGPLAN'88 Conf. on Prog. Lang. Design and Implementation, June 1988.

[NF84] A. Nicolau and J. A. Fisher, "Measuring the parallelism available for very long instruction word architectures," IEEE Trans. Comput. vol. C-33, Nov. 1984.

[Ni85] A. Nicolau, "Uniform parallelism exploitation in ordinary programs," Proceedings of the 1985 International Conference on Parallel Processing, 1985.

[Po91] R. Potasman, "Percolation-based compiling for evaluation of parallelism and hardware design trade-offs", UC Irvine, Technical report 91-80,1991.

[PoLi90] R. Potasman, J. Lis, A. Nicolau, D. Gajski, "Percolation based synthesis", Proceedings of the 27th Design Automation Conference, 1990.

[PS88] A. R. Pleszkun and G. S. Sohi, "The performance potential of multiple functional unit processors," in Proc. 15th Annu. Symp. Comput. Architecture, IEEE Computer Society Press, May 1988.

[SV87] G. S. Sohi and S. Vajapeyam, "Instruction issue logic for high-performance interruptable pipelined processors," in Proc. 14th Annu. Symp. Comput. Architecture, IEEE Computer Society Press, June 1987.

[WS84] S. Weiss and J. E. Smith, "Instruction issue logic for pipelined supercomputers," in Proc. 11th Annu. Symp. Comput. Architecture, IEEE Computer Society Press, June 1984.