Utilizing Horizontal and Vertical Parallelism with a No-Instruction-Set Compiler for Custom Datapaths

Mehrdad Reshadi, Bita Gorjiara, Daniel Gajski
Center for Embedded Computer Systems (CECS), University of California Irvine
{reshadi, bgorjiar, gajski}@cecs.uci.edu

Abstract

Performance of programs can be improved by utilizing their horizontal and vertical parallelism. In some processors (VLIW based), compiler can utilize horizontal parallelism by controlling the schedule of independent operations. Vertical parallelism is utilized through pipelining. However, in all processors, structure of pipeline is fixed and compiler has no control over it. In Application-Specific-Instruction set-Processors (ASIPs), pipeline structure can be customized and utilized in the program through custom instructions. Practical constraints on the instruction decoder limit the number and complexity of custom instructions in ASIPs. Detecting the frequent and beneficial custom instructions and incorporating them in the compiler are complex and sometimes very time consuming tasks.

In this paper, we present an architecture that does not limit the number of custom functionalities that can be implemented on its datapath. Instead of using custom instructions and then relying on the decoder in hardware to generate the control signals, we generate the control signal values in compiler. Since there are no predefined instructions in this architecture, we call it No-Instruction-Set-Computer (NISC). The NISC compiler maps the application directly on the datapath. It has complete fine grain control over datapath and hence can very well utilize resources in the hardware as well as horizontal and vertical parallelism in the program. We also explain the algorithm for mapping the CDFG of a program on a given datapath in NISC. Using our algorithm and a NISC architecture with the datapath of a MIPS, we achieved up to 70% speedup over the traditional MIPS compiler. In another experiment, we started from a base architecture and customized it by adding resources and interconnects to increase its horizontal and vertical parallelism. The algorithm achieved up to 15.5 times speedup by utilizing the available parallelism in the program and the datapath.

1. Introduction

Performance of applications can be improved by exploiting their inherent horizontal and vertical parallelism. Horizontal parallelism occurs when multiple independent operations can be executed simultaneously. Vertical parallelism occurs when different stages of a sequence of operations can be overlapped. In processors, horizontal parallelism is utilized by having multiple functional units that run in parallel and vertical parallelism is utilized through pipelining.

In VLIW processors, the compiler controls the schedule of parallel independent operations. However, in all processors, the compiler has no control over the structure of pipeline. Therefore, the vertical parallelism of the program may not be efficiently utilized.

In Application Specific Instruction-set Processors (ASIPs), structure of pipeline can be customized for an application through custom instructions. At run time, each custom instruction is decoded and executed on the corresponding custom hardware. Due to practical constrains on size and complexity of instruction decoder and custom hardware, only few custom instructions can be actually implemented in ASIPs. Therefore, only the most frequent or beneficial custom instructions are selected and implemented. Identifying the beneficial instructions, implementing an efficient decoder, and incorporating the new instructions in the compiler are complex tasks and require special expertise; which may affect the time-to-market of these processors.

In this paper, we present an architecture in which the compiler determines both the schedule of parallel independent operations (horizontal parallelism), and the logical flow of sequential operations in the pipeline (vertical parallelism). The compiler generates code *as if* each basic block of program is executed with one custom instruction. A basic block is a sequence of operations in a program that are always executed together. Ideally, for each basic block we should have one instruction that reads the inputs of basic block from a storage (e.g. register file) and computes the outputs of basic block and stores them back. The large number of basic blocks in a typical program prevents us from using an ASIP approach to achieve the above goal. In ASIP, after reading the binary of a custom instruction from memory, it is decoded into a set of control words (CWs) that control the corresponding custom datapath and executes the custom functionality. Instead of having too many custom instructions and then relying on a large instruction decoder to generate CWs in hardware; we generate the CWs in compiler by directly mapping each basic block onto the custom datapath. Therefore, the compiler can construct unlimited number of custom functionalities utilizing both horizontal and vertical parallelism of the input program.

Since there are no predefined instructions in our architecture, we call it No-Instruction-Set-Computer (NISC). A NISC compiler compiles the application directly to the datapath. It can achieve better parallelism and resource utilization than conventional instruction-set based compilers. In fact in our experiments (Section 5), on a MIPS [3] datapath the NISC compiler achieved up to 70% speedup compared to an instruction-set-based compiler. However, NISC compiler must solve a more complex problem because it must deal with all structural details of the datapath. It should combine traditional compiler techniques with high level synthesis (HLS) techniques. The core of this compiler is a scheduling and binding algorithm. Scheduling links the operations of the application to the clock cycles and datapath resources. Binding involves three subtasks: *variable binding* assigns a value to a storage; *operation binding* assigns an operation to an FU; and *interconnect binding* selects a path between two FUs, or a storage and a FU. In our algorithm, these three subtasks are done during schedule of each operation. The algorithm has to perform scheduling and binding simultaneously (see Section 3) and support features such as datapath/controller pipelining, data forwarding, operation chaining, multi-cycle and pipelined units.

In Section 2 we describe the NISC architecture and then in Sections 3 and 4 explain the basic idea and the details of the mapping algorithm. This algorithm processes the Control Data Flow Graph (CDFG) [1] of the program backward and generates a finite state machine (FSM). The results of various experiments are shown in Section 5. Section 6 reviews related works and Section 7 concludes the paper.

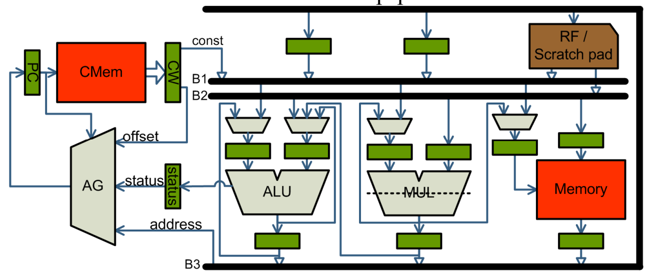


Figure 1- A sample NISC architecture.

2. NISC Architecture

While the controller of NISC has a predefined structure, its datapath can be customized to accommodate any application behavior. The datapath of NISC can be simple or as complex as datapath of a processor. The controller drives the control signals of the datapath components in each clock cycle. The NISC compiler generates the control values for each clock cycle. These values are either stored in a memory or generated via logic in the controller. Both the controller and the datapath can be pipelined. Figure 1 shows a sample NISC architecture with a memory based controller and a pipelined datapath that has partial data forwarding, multi-cycle and pipelined units, as well as data memory and register file. Note that few high level synthesis techniques can generate such complex datapaths directly from application. Such structural details are also invisible to the traditional compilers that rely on instruction abstraction.

The compiler translates each basic block to a sequence of CWs that run sequentially without interruption and execute the basic block on the datapath. In other words, any pipeline stall or context switch (e.g. interrupt routing call) happens only between basic blocks. This is analogous to traditional processors in which pipeline stalls or context switches happen between instructions.

In presence of controller pipeline (e.g. registers PC, CW and Status in Figure 1), the compiler should also make sure that the branch delay is considered correctly and is filled with other independent operations. In Sections 4.1 and 4.2 we describe the details of mapping CFG and DFG of a program to a given datapath.

The datapath can be generated (allocated) using different techniques. For example, it can be an IP, reused form other designs, or generated by high level synthesis. The program, written in a high level language such as C, is first compiled and optimized by a front end and then mapped (scheduled and bound) on the given datapath. The result is translated to an RTL hardware description that is used for simulation (validation), synthesis (implementation), etc. At any point, after compilation down to the implementation, the results can be analyzed, and the design can be improved by refining the datapath and recompiling the program on the refined datapath. This kind of flow is possible only if we can compile an application directly to a given pre-synthesized datapath. Figure 2 shows the above flow.

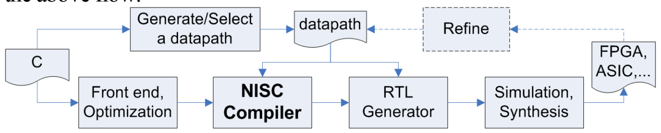


Figure 2- Generating a NISC architecture for an application.

3. Overview of the Mapping Algorithm

In this section we illustrate the basis of our scheduling and binding algorithm using an example. The input of algorithm is the CDFG of an application, netlist of datapath components and the clock period of the system. The output is an FSM in which each state represents a set of register transfers actions (RTAs) that execute in one clock cycle. An RTA can be either a data transfer through buses / multiplexers / registers, or an operation executed on a functional unit. The set of RTAs are later used to generate the control bits of components.

As opposed to traditional HSL, we can not schedule operations merely based on the delay of the functional units. The number of control steps between the schedule of an operation and its successor depends on both the binding of operations to functional units (FU) and the delay of the path between corresponding FUs. For example, suppose we want to map DFG of Figure 4 on datapath of Figure 5. Operation shift-left (>>) can read the result of operation + in two ways. If we schedule operation + on *U2* and store the result in register file *RF*, then operation >> must be scheduled on *U3* in next cycle to read the result from *RF* through bus *B2* and multiplexer *M2*. Operation >> can also be scheduled in the same cycle with operation + and read the result directly from *U2* through multiplexer *M2*. Therefore, selection of the path between *U2* and *U3* can directly affect the schedule. Since knowing the path delay between operations requires knowing the operation binding, the scheduling and binding must be performed simultaneously.

The basic idea in the algorithm is to schedule an operation and all of its predecessors together. An *output operation* in the DFG of a basic block is an operation that does not have a successor in that basic block. We start from output operations and traverse the DFG backward. Each operation is scheduled after all its successors are scheduled. The scheduling and binding of successors of an operation determine when and where the result of that operation is needed. This information can be used for: utilizing available paths between FUs efficiently, avoiding unnecessary register file read/writes, chaining operations, etc.

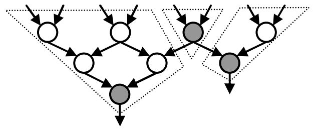
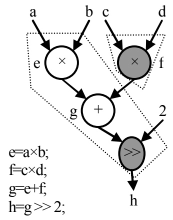


Figure 3- Partitioning a DFG into output sub-trees.

We partition the DFG of the basic block into sub-trees. The root of a sub-tree is an output operation. The leaves are input variables, constants, or output operations from other basic blocks. If the successors of an operation belong to different sub-trees, then that operation is considered as an *internal output* and will have its own sub-tree. Such nodes are detected during scheduling. Figure 3 shows an example DFG that is partitioned into three sub-trees. The roots of the sub-trees (shown with shaded nodes) are the output operations. The algorithm schedules each sub-tree separately. If during scheduling of the operations of a sub-tree, the schedule of an operation fails, then that operation is considered an internal output and becomes the root of a new sub-tree. A sub-tree is available for schedule as soon as all successor of its root (output operation) are scheduled. Available sub-trees are ordered by the mobility of their root. The algorithm starts from output nodes and schedules backward toward their inputs, therefore more critical outputs tend to be generated towards the end of the basic block (similar to ALAP schedule).

Consider the example DFG of Figure 4 to be mapped on the datapath of Figure 5. Assume that the clock period is 20 units and delays of *U1*, *U2*, *U3*, multiplexers and busses are 17, 7, 5, 1 and 3 units, respectively. We schedule the operations of basic block so that all results are available before last cycle, i.e. 0; therefore, the RTAs are scheduled in negative cycle numbers. In each step, we try to schedule the sub-trees that can generate their results before a given cycle *clk*. The *clk* starts from 0 and is decremented in each step until all sub-trees of a basic block are scheduled.



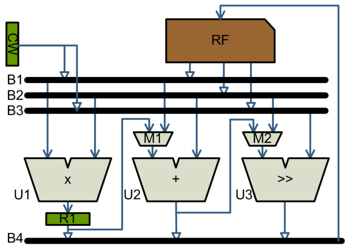


Figure 4- A sample DFG. **Figure 5- A sample datapath.**

During scheduling, different types of values may be bound to different types of storages (variable binding). For example, global variables may be bound to memory, local variables to stack or register file, and so on. A constant is bound to memory or control word (CW) register, depending on its size. A control word may have limited number of constant fields that are generated in each cycle along with the rest of control bits. These constant fields are loaded into the CW register and then transferred to a proper location in datapath. The NISC compiler determines the values of constant(s) in each cycle. It also schedules proper set of RTAs to transfer the value(s) to where it is needed.

When scheduling an output sub-tree, first step is to know where the output is stored. In our example, assume *h* is bound to register file *RF*. We must schedule operation >> so that its result can be stored in destination *RF* in cycle -1 and be available for reading in cycle 0. We first select a FU that implements >> (operation binding). Then we make sure that a path exists between selected FU and destination *RF* and all elements of the path are available (not reserved by other operations) in cycle -1 (interconnect binding). In this example we select *U3* for >> and bus *B4* for transferring the results to *RF*. Resource reservation will be finalized if the schedule of operands also succeeds. The next step is to schedule proper RTAs in order to transfer the value of *g* to the left input port of *U3* and constant *2* to the right input port of *U3*. Figure 6 shows the status of schedule after scheduling the >> operation. The figure shows the set of RTAs that are scheduled in each cycle to read or generated a value. At this point, *B3* and *M2* are considered the *destinations* to which values of *2* and *g* must be transferred in clock cycle -1, respectively.

clock→ operation↓	-3	-2	-1
g			M2=?;
2			B3=?;
h			B4=U3(M2, B3); RF(h)=B4;

Figure 6- Schedule of RTAs after scheduling >> operation.

In order to read constant *2*, we need to put the value of CW register on bus *B3*. As for variable *g*, we schedule the + operation on *U2* to perform the addition and pass the result to *U3* though multiplexer *M2*. Note that delay of reading operands of + operation and executing it on *U2*, plus the delay of reading operands of >> operation and executing it on *U3* and writing the results to *RF* is less than one clock cycle. Therefore, all of the corresponding RTAs are scheduled together in clock cycle -1. The algorithm chains the operations in this way, whenever possible. The new status of scheduled RTAs is shown in Figure 7. In the next step, we should schedule the × operations to deliver their results to the input ports of *U2*.

clock→ operation↓	-3	-2	-1
e			M1=?;
f			B2=?;
g			M2=U2(M1, B2);
2			B3=CW;
h			B4=U3(M2, B3); RF(h)=B4;

Figure 7- Schedule of RTAs after scheduling + operation.

The left operand (*e*) can be scheduled on *U1* to deliver its result through register *R1* in cycle -2 and multiplexer *M1* in cycle -1. At this point, no other multiplier is left to generate the right operand (*f*) and directly transfer it to the right input port of *U2*. Therefore, we assume that *f* is stored in the register file and try to read it from there. If the read is successful, the corresponding × operation (*f*) is considered as an internal output and will be scheduled later. Figure 8 shows the status of schedule at this time. The sub-tree of output *h* is now completely scheduled and the resource reservations can be finalized.

clock→ operation↓	-3	-2	-1
c		B1=RF(c);	
d		B2=RF(d);	
e		R1=U1(B1, B2);	M1=R1;
f			B2=RF(f);
g			M2=U2(M1, B2);
2			B3=CW;
h			B4=U3(M2, B3); RF(h)=B4;

Figure 8- Schedule of RTAs after scheduling *h* sub-tree.

The sub-tree of internal output *f* must generate its result before cycle -1 where it is read and used by operation +. Therefore, the corresponding RTAs must be scheduled in or before clock cycle -2 and write the result in register file *RF*. The path from *U1* to *RF* goes through register *R1* and hence takes more than one cycle. The second part of the path (after *R1*) is scheduled in cycle -2 and the first part (before *R1*) as well as the execution of operation × on *U1* is scheduled in cycle -3. The complete schedule is shown in Figure 9.

clock→ operation↓	-3	-2	-1
a	B1=RF(a);		
b	B2=RF(b);		
c		B1=RF(c);	
d		B2=RF(d);	
e		R1=U1(B1, B2);	M1=R1;
f	R1=U1(B1, B2);	B4=R1; RF(f)=B4;	B2=RF(f);
g			M2=U2(M1, B2);
2			B3=CW;
h			B4=U3(M2, B3); RF(h)=B4;

Figure 9- Schedule of RTAs after scheduling all sub-trees.

In the above example, we showed how the DFG is partitioned into sub-trees during scheduling. We also showed how pipelining, operation chaining, and data forwarding are supported during scheduling.

4. Compiling an application to a custom datapath

In this section we describe our algorithm for compiling the application to a custom datapath. When compiling the CDFG of each function of a program, we must consider the structure of the controller for compiling the control flow graph (CFG) and consider the structure of datapath for compiling the DFG. This process is described in the next two subsections. Description of the algorithm uses the following definitions:

- Each basic block has a schedule status *ss*, where *ss.RTAs*(*clk*) stores the set of scheduled RTAs in clock cycle *clk*, and *ss.resTable*(*clk*) stores the reservation status of resources in clock cycle *clk*, and *ss.length* shows the number of scheduled states for that block.
- For an operation *op*, *op.result* is the value generated by *op* and *op.operands* is the list of results of predecessors of *op*.
- For a functional unit *FU*, *FU.output* is the output port of *FU* and *FU.inputs* is the set of input ports of *FU*. A functional unit may implement multiple operations. For each operation, *FU.timing* represents the delay of the unit (or its stages if it is pipelined) as well as the duration of applying the control signals to the unit.
- A path *p* is the list of resources that can transfer a value from one point to another. These resources include busses, multiplexers and registers. The timing of resources of *p* is stored in *p.timings* and is calculated base on delay of buses or multiplexers, or setup time and read delay of registers or register-files.
- A destination *dst* is a storage or an input port of a functional unit.

4.1 Mapping the CFG of the program

The result of NISC compiler is an FSM that can be implemented in logic or using a memory. In a memory-based implementation the state register is a program counter register (PC). Therefore, a state change in the FSM corresponds to incrementing the PC or loading it with a new value using a jump operation. While incrementing PC always takes one cycle, loading it with a new value may take more than one cycle. The result of scheduling a basic block is always a sequence of states (marked by value of *clk*). We may only need a jump at the end of a basic block, if the last state of the block is not before the first state of the next basic block. In the algorithm, we assume that the order of basic blocks is given, and that there may be jump operation at the end of some basic blocks.

Since we perform the scheduling backward, the result will be a set of states numbered from –*N* to +*bd*. The return address of a function is loaded into PC at state 0. Constant *bd* is the branch delay of the architecture, i.e. in a basic block, after loading the target address of a jump operation into PC, *bd* more control words will be executed from that basic block. Value of *bd* depends on the distance between *PC* and control word register, which is fixed and unique. Usually, this delay is 0 or 1 cycle in NISC.

```
00 ScheduleFunction(FSM fsm, ordered list of basic blocks blkList)
01   clk = 0;
02   bd = branch delay;
03   foreach (blk ∈ reverse of blkList)
04     if (blk has a jump operation)
05       ScheduleOperation(blk.jump, clk, blk.ss, PC);
06     ScheduleBasicBlock(blk, clk+bd);
07     add blk.ss states to fsm;
08     clk = clk – blk.ss.length;
```

Figure 10- The ScheduleFunction procedure.

In procedure *ScheduleFunction* (Figure 10), the *blkList* contains the topologically ordered list of basic blocks where the last element of the list is the *return block*. The blocks of *blkList* are processed in reverse order, starting from return block and after scheduling each block, the results are added to the *fsm*. In the main loop of *ScheduleFunction* (lines 3-8), before scheduling the body of a basic block, the jump operation at the end of block is scheduled. The same way that a + operation is mapped to an adder or ALU and writes its results to a register or register file, the jump is considered an operation that is mapped to address generator and writes its result to the PC register in cycle *clk*. In this way, we can schedule jump the same way that we schedule other operations (line 5). In order to make sure that the branch delay of the jump operation is filled by other operations in the basic block, we try to schedule the DFG of the basic block from cycle *clk*+*bd* (line 6). After scheduling each basic block, the new value of *clk* is calculated by decrementing the number of states in the block (line 7). The *ScheduleBasicBlock* and *ScheduleOpertion* functions are described in Section 4.2. After scheduling all functions of a program, *fsm* will contain the final FSM of the design.

4.2 Mapping the DFG of the program

The variable, operation, and interconnect bindings are performed during the schedule of each operation. We also allow pre-binding of variables and operations so that the designer or other algorithms can control the results. For example, a partitioning algorithm may partition the variables and pre-bind them to two memory units.

```
00 ScheduleBasicBlock(block blk, clock lastClock)
01    Roots = {output operations in blk.DAG};
02    clk = lastClock;
03    while(Roots ≠ ∅)
04       AvailableOutputs = ∅;
05       foreach (operation op ∈ Root)
06          if (all successor of op are scheduled after clock clk)
07             AvailableOutputs = AvailableOutputs + {op};
08       Sort AvailableOutputs by OperationPriorities;
09       foreach (operation op ∈ AvailableOutputs)
10          internalOutputs=∅;
11          if (op.result is not pre-bound to a storage)
12             bind op.result
13          destination dst = storage of op.result
14          if ( ScheduleOperation(op, clock ,blk.ss, dst))
15             Roots = Roots – {op} + internalOutputs;
16       clk=clk-1;
```

Figure 11- The ScheduleBasicBlock procedure.

Figure 11 shows the *ScheduleBasicBlock* procedure that performs the scheduling and binding for each basic block of a CDFG. In the main loop of this function (lines 3-16) the *available* output operations, i.e. sub-tree roots that can generate their results at clock cycle *clk*, are collected and sorted based on a priority function, such as operation mobility. During scheduling of each of these output operations, some internal outputs may be generated. If the schedule of the operation is successful, then the operation is removed from the set of sub-tree roots (*Roots*) and the newly generated internal outputs are added to the list in order to be processed later (lines 14-15). In each iteration of the loop, the *clk* is decremented and available output operations are collected and scheduled until all sub-trees in the block are processed.

```
00bool ScheduleOperation(operation op, clock clk, schedule status ss, destination dst)
01  if (op is pre-bound to a functional unit)
02     F = {functional unit to which op is pre-bound};
03  else
04     F = {functional units that implement op sorted by UnitPriorities};
05  foreach(FU ∈ F)
06     P = {paths from FU.output to dst sorted by PathPriorities};
07     foreach(p ∈ P)
08        p.timings.end = clock;
09        calculate p.timings.start;
10        if (resources of p are not reserved in ss.resTable)
11           FU.timing.end = p.timings.start;
12           calculate FU.timing.start;
13           if (FU is not reserved in ss.resTable)
14              copyStatus = ss;
15              if (ScheduleOperands(op, FU.timing.start, copyStatus, FU))
16                 ss = copyStatus;
17                 reserve FU and p in ss.resTable;
18                 add corresponding RTAs to ss.RTAs;
19                 return TRUE;
20  bind op.result;
21  if (ScheduleRead(op.result, clk, ss, dst))
22     internalOutputs = internalOutputs + {op};
23     return TRUE;
24  return FALSE;
```

Figure 12- The ScheduleOperation function.

The *ScheduleOperation* function (Figure 12) tries to schedule an operation *op* so that its result is available at *dst* at clock cycle *clk*. If *op* is not pre-bound to a specific functional unit, then the list of functional units that can execute *op* is stored in *F* and sorted by the UnitPriorities (lines 1-4). This priority function depends on the delay of the unit as well as the paths from output of the unit to the destination *dst*. After selecting a functional unit *FU*, all paths from *FU* to *dst* are stored in *P* and sorted by a PathPriority. The timings of *FU* and a selected path *p* are calculated so that the output of *FU* is available at *dst* at clock cycle *clk* (lines 7-12). If *FU* and all of the resources on the path *p* are not reserved in the *ss.resTable* at the corresponding calculated times, then algorithm tries to schedule the operands of *op* by calling the *ScheduleOperands* function. If the schedule of operands succeeds, then selected functional unit *FU* and path *p* are reserved (operation and interconnect binding) (lines 15-19). We pass a copy of scheduling status (*copyStatus*) to function *ScheduleOperands* to make sure that original status changes only if all operands are successfully scheduled. If scheduling failed after trying all functional units, the *ScheduleOperation* function tries to bind the result of operation to a storage and schedule a read from that storage. If the read succeeds, the operation is added to the *internalOutputs* for later processing.

```
00 bool ScheduleOperands(operation op, clock clk, schedule status ss, functional
01 unit FU)
02    foreach(operand o ∈ op.operands)
03       destination dst = FU.inputs corresponding to o;
04       if (o is a variable or a constant)
05          if (o is not pre-bound to a storage)
06             bind o to a storage;
07          if (! ScheduleRead(o, clk, ss, dst))
08             return FALSE;
09       else if (! ScheduleOperation(o, clk, ss, dst))
10             return FALSE;
11    return TRUE;
```

Figure 13- The ScheduleOperands function.

The *ScheduleOperands* function (Figure 13) schedules the operands of an operation *op* on a selected functional unit *FU* so that their values are available on corresponding input ports of *FU* at clock cycle *clk*. If an operand is a variable or a constant, then this function tries to schedule a read from the corresponding storage. Otherwise, it calls the *ScheduleOperation* function. The function succeeds only if all operands can be scheduled.

```
00 bool ScheduleRead(value v, clock clk, schedule status ss, destination dst)
01    P = {paths from storage of v to dst sorted by PathPriorities};
02    foreach(p ∈ P)
03       p.timings.end = clk;
04       calculate p.timings.start;
05       if (resources of p not reserved in ss.resTable)
06          reserve p in ss.resTable;
07          add corresponding RTAs to ss.RTAs
08          return TRUE;
09    return FALSE;
```

Figure 14- The ScheduleRead function.

In the *ScheduleRead* function (Figure 14), the best available path that can transfer a value from its storage to the specified destination at clock cycle *clk* is selected and scheduled.

5. Experiments

In this section we report preliminary results of implementing our algorithm in a NISC compiler that is being developed as part of the NISC based design tool set. The input to the compiler is the netlist of datapath components as well as the application written in ANSI C. To evaluate our algorithm we compiled a set of benchmarks on a set of architectures and evaluated the schedules. We reused the same datapath to compile and implement different benchmarks. We also started from a simple architecture and iteratively refined it to improve the result performances. For each experiment, we simulated and synthesized the generated RTL description in order to extract the timing information.

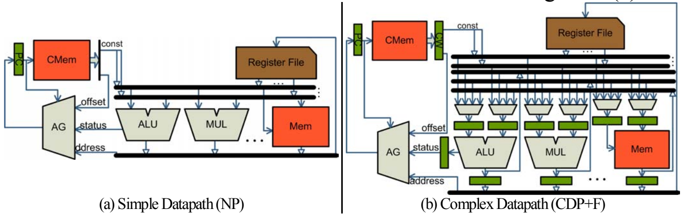
For benchmarks, we used the *bdist2* function (from MPEG2 encoder), *DCT* 8x8, *FFT*, and a *sort* function (implementing the bubble sort algorithm). The *FFT* and *DCT* benchmarks have data independent control graphs. The *bdist2* benchmark works on a 16×h block and we used h=10 in our experiments. For the *sort* benchmark, we calculated the best case and worst case results for sorting 100 elements. Among these benchmarks, *FFT* has the most parallelism and *sort* is a fully sequential code. A demo of the tool and the details of benchmarks and architectures are available at [2].

First, we evaluated the schedule of benchmarks on two processor-like NISC architectures. The datapath of NM1 architecture is the same as a MIPS M4K Core [3]. The NM2 architecture extends the datapath of NM1 by adding one more ALU and 2 more register-file read ports. Because of their similar datapath, the clock periods of these architectures are similar. For MIPS, NM1, and NM2, Table 1 shows the execution cycle counts of benchmarks and their corresponding speedups vs. MIPS. We used a gcc-based cross compiler to compile and optimize the benchmarks for MIPS. Note that although NM1 and MIPS have the same datapath, the benchmarks run up to 70% faster on NM1. The parallelism in NM1 (and MIPS) is limited by the number of register-file read/write ports. However, our algorithm has well utilized the pipelining and data forwarding paths between components and achieved the speedup by avoiding unnecessary accesses to the register file and controlling the flow of information in the pipeline. By utilizing the vertical and horizontal parallelism in NM2, the benchmarks run up to 100% faster than MIPS. These results show that by having more control over datapath, the NISC compiler generates better results than an instruction-set-based compiler.

Table 1- Cycle counts and speedups on MIPS and MIPS-like NISCs.

	Cycle count			Speedup vs. MIPS		
	MIPS	**NM1**	**NM2**	**MIPS**	**NM1**	**NM2**
bdist2: block 16x10	6727	5204	4363	1.00	1.29	1.54
DCT 8x8	13058	10772	10644	1.00	1.21	1.23
FFT	277	162	133	1.00	1.71	2.08
Sort: Best case (N=100)	45642	40103	40004	1.00	1.14	1.14
Sort: Worst case (N=100)	50493	54656	54557	1.00	0.92	0.93

To evaluate the utilization of vertical parallelism, we used a set of architectures that had the same number and type of functional units and storages but had different pipeline structure. We started with an architecture with no pipelining (NP) similar to Figure 15(a). Then we added controller pipelining (CP) by adding *CW* and *status* registers in front of control memory (CMem) and address generator (*AG*), respectively. We then added datapath pipelining (CDP) by adding registers to the input/output ports of functional units and data memory. At the end, we added data forwarding (CDP+F) by adding interconnects from output of functional units to the input registers of other functional units. The final architecture is similar to what is shown in Figure 15(b).



(a) Simple Datapath (NP) (b) Complex Datapath (CDP+F)

Figure 15- Experimental NISC Datapaths

After compiling the benchmarks on the above datapaths, we generated corresponding Verilog files. To get the number of execution cycles, we simulated the files; and to get the clock periods we synthesized them on a Xilinx Virtex-II Pro FPGA package using the Xilinx ISE tools. In Table 2 under each architecture column, the clock period; as well as number of cycles (first column); execution delay (cycles count × cycle period) in micro-seconds; and speedup vs. NP is shown. Note that while adding pipelining reduces the clock period, it may increase the cycle counts especially if there is not enough parallelism in the benchmark. Therefore, except for *FFT*, the cycle count of other benchmarks increases when we move from NP, to CP and CDP. The considerable decrease of execution cycle counts from CDP to CDP+F is because of utilizing the data forwarding paths. Nevertheless, the execution times of benchmarks have decreased due to improvements of cycle periods.

	NP			**CP**			**CDP**			**CDP+F**			**NM1**			**NM2**		
clock (ns)	12.4			9.8			5.4			6.7			8.6			8.7		
bdist2	6143	76.2	1.0	6326	62.0	1.2	7168	38.7	2.0	5226	35.0	2.2	5204	44.8	1.7	4363	38.0	2.0
DCT	10450	129.6	1.0	11764	115.2	1.1	14292	77.2	1.7	13140	88.0	1.5	10772	92.6	1.4	10644	92.6	1.4
FFT	219	2.7	1.0	220	2.2	1.3	218	1.2	2.3	166	1.1	2.4	162	1.4	1.9	133	1.2	2.3
Sort: Best	25447	315.5	1.0	35349	346.4	0.9	84161	454.5	0.7	74162	496.9	0.6	40103	344.9	0.9	40004	348.0	0.9
Sort: Worst	35149	435.8	1.0	49902	489.0	0.9	98714	533.1	0.8	88715	594.4	0.7	54656	470.0	0.9	54557	474.6	0.9

Table 2- Execution delay (us) of benchmarks and speedup vs. NP.

In the previous experiments, we neither used any optimization (such as loop unrolling) nor modified the source code of benchmarks to increase the parallelism. As another set of experiments, we partially unrolled and combined the nested loops in the source code of *DCT*, and scheduled the new version on our architectures. We also designed a custom architecture for the modified *DCT* and collected the results after simulation and synthesis. Figure 16 shows the main loop of the new version. In the original version, there were three loops that iterate from 0 to 7 for three variables *i*, *j*, and *k*. We unrolled the inner loop (*k*) and combined the loops of *i* and *j* into one loop. The commented codes in Figure 16 show how the conversion has been done. In this figure, *(*x*) means that the memory content at address *x* is being loaded or stored.

```
ij=0;
do {
   //Original: sum+= A[i][k] × B[k][j];
   //Converted to: sum+= *(A+ (i×8+k) ) × *(B + (k×8+j) );
   //Converted to: sum+= *(A+ ((ij&0xF8)|k) ) × *(B + (k|(ij&0x7)) );
   i8 = ij & 0xF8; //i × 8 => (i8|k) = (i×8+k)
   j = ij & 0x7;
   aL = *(A+ (i8|0) ); bL = *(B + (0|j) );  sum =  aL × bL;
   aL = *(A+ (i8|1) ); bL = *(B + (8|j) );  sum += aL × bL;
   aL = *(A+ (i8|2) ); bL = *(B + (16|j) ); sum += aL × bL;
   aL = *(A+ (i8|3) ); bL = *(B + (24|j) ); sum += aL × bL;
   aL = *(A+ (i8|4) ); bL = *(B + (32|j) ); sum += aL × bL;
   aL = *(A+ (i8|5) ); bL = *(B + (40|j) ); sum += aL × bL;
   aL = *(A+ (i8|6) ); bL = *(B + (48|j) ); sum += aL × bL;
   aL = *(A+ (i8|7) ); bL = *(B + (56|j) ); *(C + ij) = sum + (aL × bL);
   ++ij;
} while(ij<64);
```

Figure 16- The main loop of modified DCT.

Figure 17 shows the custom NISC architectures that we designed for the modified DCT. The architecture has very irregular deep pipelines, and the compiler should perform operation chaining on the *OR* and *ALU* components. The multiply and add (accumulate) operations are chained as well. We used the pre-binding feature of our algorithm and forced it to bind variables *aL*, *bL* and *sum* in Figure 16 to the registers *aL*, *bL* and *Sum* in datapaths of Figure 17. Table 3 shows cycle periods, cycle counts, execution delays and speedups vs. architecture NP for both the DCT and its modified version. Since the modified DCT has inherent horizontal and vertical parallelism, the added interconnects and resources to NP improve the performance significantly in other architectures (upto 15.5 times in the Custom architecture). The results also show that, although the modified DCT runs slower than the original DCT on previous architectures, it outperforms the others on the custom datapaths.

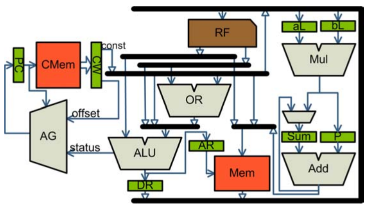


Figure 17- Custom NISC for DCT.

arch.	**clock (ns)**	**Original DCT**			**Modified DCT**		
		#clocks	delay(µs)	speedup	#clocks	delay(µs)	speedup
NP	12.4	10450	129.6	1.0	33186	411.6	1.0
CP	9.8	11764	115.2	1.1	24604	241.2	1.7
CDP	5.4	14292	77.2	1.7	38850	209.8	2.0
CDPF	6.7	13140	88	1.5	17434	116.8	3.5
NM1	8.6	10772	92.6	1.4	13586	116.8	3.5
NM2	8.7	10644	92.6	1.4	11154	97	4.2
Custom	9	-	-	-	2952	26.6	15.5

Table 3- Performances.

Note that in these examples, we only customized the datapath and the compiler automatically utilized the custom pipeline structure. We did not need to generate any custom instruction and modify the compiler to use it. Although in its early experimental phases, our NISC compiler is generating encouraging results on different architecture ranging from simple to complex. The results indicate that different architectural features such as controller / datapath pipelining, data forwarding, and operation chaining, are very well utilized by our algorithm.

6. Related works

Because the architecture style of NISC is new, little research has been done on the mapping algorithms for NISC. However, some of techniques developed in the areas of ASIPs, high level synthesis, and retargetable compilers can be directly or indirectly related to NISC. The application analysis techniques in ASIP domain can be used to select or generate proper datapaths in NISC. A survey of different ASIP techniques and approaches is presented in [4]. In this paper we presented a scheduling and binding algorithm for mapping the CDFG of the program on a given datapath in NISC. There has been an extensive body of work on scheduling and binding algorithms in the area of high level synthesis and retargetable compilers, which we review in this section.

List-based scheduling techniques [5] are used to solve resource constrained scheduling problem in which the number of resources of different types are limited. List scheduling processes each control step sequentially. At each control step, it tries to choose the best operation from the list of candidate operations, subject to resource constraints. List scheduling uses a ready-list, which keeps all nodes that their predecessors are already scheduled. The ready-list is always sorted with respect to a priority function. The priority function always resolves the resource contention among operations, i.e. operations with lower priority will be deferred to the next or later control steps. The quality of the results produced by a list-based scheduler depends predominantly on its priority function.

Mobility of the operation, i.e. the difference between ASAP (as soon as possible) and ALAP (as late as possible) times, is commonly used as the priority function in many HLS systems. Different priority functions and heuristics have been proposed to improve the quality of list scheduling. The proposed list scheduling algorithms in [6] and [7] uses mobility as the primary priority functions. To break the tie among a set of available operations with similar mobility, they assign higher priority to those operations that contribute to the same output. Before scheduling begins, they analyze the outputs of operations in the DFG by constructing a set of trees (cones) that start from output nodes as roots. However, they use a conventional scheduler that starts from inputs and proceeds forward, and the output trees are only used to break the tie during schedule. A similar approach is used in [8] and [9] for scheduling on VLIW architectures. Output trees in DFG are also used for instruction selection using the maximal-munch algorithm. Processing the DFG backward, from outputs towards inputs, has proven to be very fruitful. However, this idea has been mainly used in priority functions but not the scheduling algorithm itself.

Getting a fixed architecture model as input is a common assumption in retargetable compilers, mostly used for ASIPs. But usually in these compilers the architecture model is described in terms of instructions, which is a much higher level of abstraction than the structural details of the architecture. Even compilers such as RECORD [10] and CHESS [11] that use a structural description of architecture, extract the higher level instruction information for using in the compiler. The RECORD compiler extracts behavioral model of instructions from MIMOLA HDL [12]. They assume a horizontal microcode machine with single cycle operation. They process the structure of the datapath from destination storages towards source storages to extract valid register transfers (RTs). After analyzing the controller, they reject illegal RTs that do not correspond to an instruction, and use the remaining RTs in the compiler. The CHESS compiler uses the nML language [13] to extract the instruction set graph (ISG) that captures structural resources in the architecture that are used by each instruction.

Regardless of the approaches, every compiler generates a stream of processor instructions and assumes that the processor itself deals with the control signals of its component. Since there is no instruction in NISC, the compiler directly maps the program to the datapath. In this way, compiler has complete fine-grain control over datapath and can achieve better parallelism and resource utilization. However, not only the compiler should generate the schedule, it should also generate the control values of architecture component in each cycle. Therefore, the NISC compiler must deal with much more structural details and solve a more complex problem than traditional processor compilers.

In all HLS approaches scheduling is done mainly based on the delay of functional units, while all or part of binding (especially interconnect binding) is done afterwards. This is not possible in NISC and scheduling and binding *must* be done simultaneously (see Section 3).

7. Conclusion

In this paper, we introduced No-Instruction-Set-Computer (NISC) architecture and described an algorithm for compiling applications on this architecture. In NISC, there is no predefined instruction. The compiler maps the program directly on the datapath and generates the control signal values that execute the program. Since the compiler has complete control over datapath, it can construct any custom functionality and utilize horizontal and vertical parallelism in programs. The NISC approach also enables design reuse and refinement. We showed that a NISC with a datapath similar to that of a MIPS M4K can perform up to 70% better. We predict the same applies when using datapath of other embedded processor cores.

Our algorithm is different from HLS techniques because it assumes that the datapath is given and is fixed during scheduling and binding. It performs the scheduling and binding simultaneously while processing the CDFG backward. It is also different from conventional instruction-set based compiler techniques because it directly maps the program on a given datapath without using any high-level instruction abstraction. Consequently, it must deal with all structural details of the architecture and solve more complex problems.

Our experiments indicate that the compiler efficiently supports features such as controller / datapath pipelining, data forwarding, multi-cycle and pipeline units, and operation chaining. In one set of experiments, we partially unrolled and combined the nested loops in the source code of a DCT 8x8, and compiled it on a custom datapath. We achieved considerably better results in terms of cycle count and total execution delay (up to 15.5 times faster). The results of our experiments indicate that in presence of parallelism in the application and the datapath, our algorithm generates promising results on the NISC architecture.

8. References

[1] A. Orailoglu and D.D. Gajski, “Flow graph representation”, Design Automation Conference, 1986.

[2] http://www.cecs.uci.edu/~reshadi/projects/nisc/

[3] MIPS32® M4K™ Core, http://www.mips.com

[4] M.K. Jain, M. Balakrishnan, and A. Kumar, “ASIP Design Methodologies: Survey and Issues”, In Proceedings of the Fourteenth International Conference on VLSI Design, 2001.

[5] D. Gajski, N. Dutt, A. Wu, S. Lin, "High-Level Synthesis Introduction to Chip and System Design", Kluwer Academic Publishers, 1994.

[6] S. Govindarajan, R. Vemuri, "Cone-Based Clustering Heuristic for List-Scheduling Algorithms", Proceedings of European Design & Test Conference (ED&TC), 1997.

[7] A.M. Sllame, V. Drabek, "An efficient list-based scheduling algorithm for high-level synthesis", Proceedings of the Euromicro Symposium on Digital System Design, 2002.

[8] E. Ozer, S. Banerjia, “Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures”, MICRO-31, 1998.

[9] J. R. Ellis, “Bulldog: A compiler for VLIW architectures”, Cambridge, MA: The MIT Press, 1986.

[10] R. Leupers, P. Marwedel, "Retargetable Generation of Code Selectors from HDL Processor Models", European Design and Test, 1997.

[11] J. Van Praet, D. Lanneer, G. Goossens, W. Geurts, H. De Man, "A Graph Based Processor Model for Retargetable Code Generation", European Design and Test Conference, 1996.

[12] P. Marwdedel, “The MIMOLA Design System: Tools for the Design of Digital Processors”, Design Automation Conference, 1984.

[13] A. Fauth, J. Van Praet, M. Freericks, "Describing instruction set processors using nML", European Design and Test Conference, 1995.