Effective Compilation Support for Variable Instruction Set Architecture

Jack Liu, Timothy Kong, Fred Chow
Cognigine Corporation
6120 Stevenson Blvd.
Fremont, CA 94538, USA
{jackl,timk,fredc}@cognigine.com

Abstract

Traditional compilers perform their code generation tasks based on a fixed, pre-determined instruction set. This paper describes the implementation of a compiler that determines the best instruction set to use for a given program and generates efficient code sequence based on it. We first give an overview of the VISC Architecture pioneered at Cognigine that exemplifies a Variable Instruction Set Architecture. We then present three compilation techniques that, when combined, enable us to provide effective compilation and optimization support for such an architecture. The first technique involves the use of an abstract operation representation that enables the code generator to optimize towards the core architecture of the processor without committing to any specific instruction format. The second technique uses an enumeration approach to scheduling that yields near-optimal instruction schedules while still adhering to the irregular constraints imposed by the architecture. We then derive the dictionary and the instruction output based on this schedule. The third technique superimposes dictionary re-use on the enumeration algorithm to provide trade-off between program performance and dictionary budget. This enables us to make maximal use of the dictionary space without exceeding its limit. Finally, we provide measurements to show the effectiveness of these techniques.

Keywords: configurable code generation, dictionary, embedded processor, enumeration, instruction scheduling, program representation, resource modeling, variable instruction set.

1. Introduction

In recent years, processors with configurable instruction sets are becoming more and more widely used, due to the never ending effort to provide ever greater performance for running embedded applications. these application specific instruction set processors (ASIPs) [1, 2] use instruction sets customized towards a specific type of application so they can deliver efficient run-time performance for typical programs written for that application area. Because the instruction set is pre-determined, the compiler is built and configured to generate code based on a custom, fixed instruction set [16].

The Variable Instruction Set Communications Architecture (VISC Architecture^{TM}) from Cognigine represents a very different approach in the attempt to provide greater configurability in compiling embedded software. The VISC Architecture can perform a complex set of instructions consisting of multiple, fine and coarse grain operations that operate on multiple operands at the same time in one fixed hardware implementation. It also provides the capability for the compiler to tailor the instruction set to the program being compiled. In this situation, the instruction set the compiler chooses to use in compiling program A may be different from the instruction set it chooses to use for program B. In addition to coming up with the most efficient instruction sequence, the compiler also takes on the additional task of determining the best instruction set from which to derive the instruction sequence, based on its analysis of the contents of the program being compiled. This amounts to adding another dimension to the compilation problem space. Under this scenario, the design of the compiler is key to exploiting the performance advantages made possible by the VISC Architecture.

In this paper, we address the challenges in designing such a compiler for the VISC Architecture. The rest of this paper is organized as follows. Section 2 gives a brief survey of prior related work. In Section 3, the Cognigine VISC Architecture and its implementation in the CGN16000^{TM} family of network processing devices are described in more detail. In Section 4, we describe how using an abstract operation representation allows the compiler's code generator to optimize towards the core target architecture without committing to any specific instruction format. In Section 5, we present our implementation of the IDF phase that determines the final instruction set and the generated code sequences while adhering to various constraints in the hardware. In Section 6, we describe our extension to the IDF framework to promote dictionary re-use and to support trade-off between program performance and dictionary budget. In Section 7, we assess the effectiveness of these techniques via data collected on the compilation of a set of application programs. We conclude in Section 8 and point out areas of ongoing and future work.

2. Prior Work

Most prior work related to compile-time-configured instruction set architecture has been limited to the research arena. Most of the research efforts were focused on the architectural aspects, and there are little published work that addresses the compilation techniques. In his Ph.D. thesis, Talla [15] provides a good survey of the major reconfigurable computing research efforts. He defines Dynamic Instruction Set Architectures (DISA) as "a class of microprocessor architectures that provide an interface (a base ISA) that allows higher level software (such as the running program or the compiler) to extend the base ISA with additional instructions." He then defines Adaptive Instruction Level Parallel (AILP) architectures as DISA that incorporate superscalar or VLIW characteristics. Talla's work focused on a member of the AILP space called Adaptive Explicitly Parallel Instruction Computing (AEPIC) architectures. He presented a basic compilation framework that involves analyzing the program to identify portions of the code that might benefit from execution via configured hardware, followed by an instruction synthesis phase that generates "custom operations" and updates the machine description with the synthesized instructions. Subsequent phases of their compilation process are not much different from the back-end phases of typical ILP compilers. Their framework to supporting compile-time-configured instruction sets is totally different from ours, because we do not introduce synthesized instructions until the very end of the compilation. Most of our efforts are concentrated on the code generation phases of the compiler, and our approach does not require any modification to the earlier compilation phases.

3. VISC Architecture

The CGN16000 is a family of single-chip network processing devices produced by Cognigine [3]. The latest chip, the CGN16100, consists of sixteen interconnected Reconfigurable Computation Units (RCUs), each capable of supporting up to four threads. Each RCU implements the VISC Architecture, and can be regarded as a processing element for packets in the network. A program compiled from

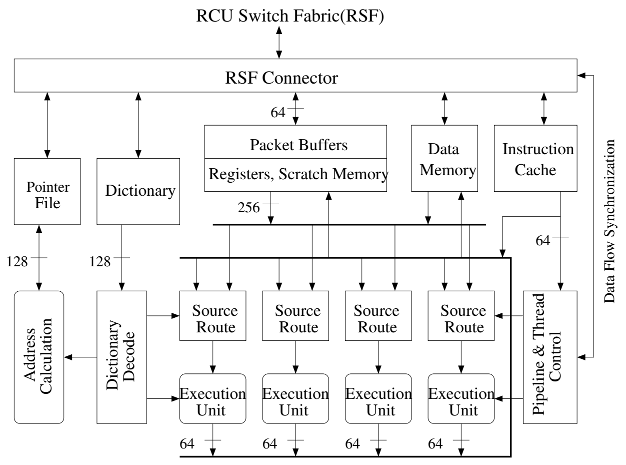


Figure 1. RCU architecture.

C can be loaded to run on any individual RCU. The rest of this paper will focus on a single RCU. We'll deal with the multi-processing and multi-thread aspects of the chip in a separate setting.

VISC represents a brand-new approach to processor architecture. The RCU incorporates four execution units, each of which is capable of performing either two concurrent 32-bit or one 64-bit arithmetic, logical or bit manipulation operation. There is memory space internal to the RCU that serves as instruction cache, data cache and dictionary store. Figure 1 gives the block level structure of the RCU in the CGN16100.

3.1. Instructions

Instructions for the RCU are either 32-bit or 64-bit long. Each CPU cycle fetches and executes one 64-bit instruction or two adjacent 32-bit instructions. In the CGN16100 implementation, each instruction has an 8-bit opcode field that indexes an entry in the dictionary. The rest of the instruction is for specifying up to four operands for the operations defined in the indexed dictionary entry (Figure 2). The operands can be encoded as either memory operands accessed via some addressing mode or immediates stored in the instruction. Of the four operands specified, two of them can also be targets for writing the results of the operations. The four memory operands have access to the memory via one of four ports according to the operands' positions in the instruction. The four ports are referred to as *port0, port1, port2* and *port3*. Each instruction contains room to store at most two immediates.

The RCU does not need to provide a separate register file, because it can access the memory space internal to the RCU without delay in performing each operation. Thus, the architecture does not require any load or store operation. The basic addressing mode is the pointer-based ad-

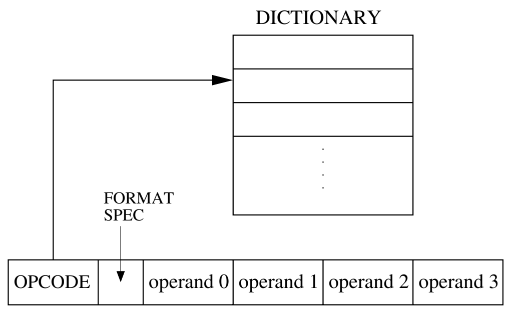


Figure 2. General form of a variable instruction

dressing mode, in which the addressed location is given by a pointer register[1] plus an offset. In addition, there are 256 locations that can be addressed directly, without involving a pointer register. This form of addressing can be regarded as the register addressing mode, and the 256 locations can be viewed logically as the register file, though they can also be indirectly addressed.

There are a number of instructions specified as fixed instructions because they are needed by all programs. A small section of the dictionary is hard-coded to support these fixed instructions. These instructions perform branches, calls, DMA[2] and thread control.

3.2. Dictionary

The dictionary is the key component of the VISC Architecture. It contains a fixed number of entries of predetermined lengths. Because CGN16100 uses an 8-bit opcode field in the instruction, the number of entries in the dictionary is fixed at 256. The content of each dictionary entry is used to specify from one to eight different operations. Each dictionary entry can be up to 128 bits long. The 128 bits are partitioned into four segments. Each of these segments can specify up to two operations to be performed in each of the four execution units by encoding the following information:

1. Operation performed.
2. Fetch path for first operand.
3. Fetch path for second operand.
4. Store path for result.

Apart from specifying the type of operation to be performed, the encoded operation also configures the execution units to work with different sizes of operands and results (8-bit, 16-bit, 32-bit and 64-bit), different signedness and different shapes of vector[3]. There are 26 different types of operations supported. When the configuration is taken into account, the number of possible operations exceeds a thousand.

The fetch and store paths for the operands and result are specified by encoding one of the following possibilities:

1. Memory operands accessed via one of the four ports, referred to as $port0$, $port1$, $port2$ and $port3$.
2. First or second immediate stored in the instruction.
3. First or second immediate stored in the dictionary entry.
4. One of the eight *transients* registers.

The transients are the special registers that hold the outputs of the eight execution units. They can be used to overcome the imbalance between the four operands provided in the instruction and the eight operations that can be performed. They are called transients because their values are overwritten each time the execution units perform operations. The transients are the cheapest to address, since they do not occupy bits in the instruction. They are best used for holding the intermediate results in a series of computations. (In the example of Figure 4(d), $trans0$, $trans1$ and $trans2$ are transients.)

An operand can be shared (i.e. used multiple times) among the different operations specified in the same dictionary entry. A memory operand can also be both source and target in the same instruction.

The dictionary can be viewed as serving two important functions. First, it is clear from the above description that the core architecture has the capability to support a myriad of operations, and each of these operations can fetch its operands and write its results in a variety of ways. Encoding all of these forms of operations in the instruction will take up a large part of the instruction space, and is not feasible in the embedded processing environment where code density is very important. By introducing the dictionary space to store the static templates for the exact forms of operations to be used, the saved space in the instruction can be used more effectively for providing operands during execution. Second, since multiple operations can be specified in a dictionary entry, this provides a new way to achieve instruction level parallelism without requiring the complex instruction fetch and decoding circuitries inherent in superscalar implementations.

3.3. VISC as Compilation Target

Traditionally, compilers play the important role of providing the means for programmers in high level languages

[1]There are 15 locations specially designated as pointer registers.

[2]DMA (direct memory access) is the only means of communications between each RCU and the outside world.

[3]The vector shapes supported are 32v8, 32v16, 64v8, 64v16 and 64v32, where the number before the 'v' is the total bit size of the vector and the number after the 'v' is the bit size of each vector element

to get at the capability of the underlying machines. The enormous capability of the RCU architecture has made it even more crucial for the compiler to live up to this role. In our case, the compiler plays the dual role of determining the best operation templates to define in the dictionary, and generating the best instruction code sequence based on these templates. In the process of doing this, the compiler also has to cater to various parameters exposed by the hardware:

1. Dictionary limit — The number of entries in the dictionary is fixed based on the size of the opcode field in the instruction.[4] There is greater chance of hitting this limit as larger programs are compiled, because they contain more operation variants.

2. Number of operands — Each instruction can provide at most four operands. Additional operands can be in the form of the transient registers, but their values are only available for one cycle. Using transients often incurs additional move instructions, which degrades performance.

3. Number of ports — In the current implementation, each instruction can fetch only four operands from memory and store two results to memory. There is also a portion of the data cache that has only a one-port connection, allowing only one access per instruction. We designate this data area as *singleport* and the rest of the data area as *multiport*.

4. Execution units — Of the four execution units in the current implementation, two are for handling arithmetic and logical operations, and two are for handling bit manipulation operations. Each dictionary entry is divided into segments, and there are specific mappings from segments to the execution units that they control.

5. Instruction encoding — Because the instructions are of fixed widths (either 32- or 64-bit), there are restrictions as to what combination of operands can be encoded in each instruction. In general, four memory operands can be specified in a 64-bit instruction, of which two of the operands can be replaced by two 16-bit immediates or one 32-bit immediate. The *format spec* field in the instruction indicates how the operands are to be decoded at run-time (see Figure 2).

6. Dictionary operation encoding — The dictionary entries can be of different sizes: 32 bits, 64 bits or 128 bits. Operations are divided into two general classes: *basic* and *extended*. A basic operation requires only 16 bits to be encoded, whereas an extended operation requires 32 bits to be encoded. Thus, a 128-bit dictionary entry can encode the maximum of eight operations only if all eight operations are basic. In addition, the types of operands allowed are not symmetric between the first and second operands of each operation, due to the limited number of bits used in the encoding.

[4]In the current implementation, the dictionary is fixed throughout the entire program execution. In future, the mechanism will be provided to allow dynamic reloading of dictionary entries so that different parts of the program can have their own dictionaries. There is also the potential to increase the size of the opcode field so it can index more dictionary entries.

The above parameters are not inherent to the architecture, and they will likely change in future implementations. But they become constraints to the compiler as it strives to improve the generated code sequences. At Cognigine, we have developed three separate techniques to tackle the compilation problems. We describe these techniques in the following sections.

4. Abstract Operation Representation

The Cognigine C Compiler is implemented by retargeting the SGI Pro64 Compiler [8]. Pro64 was evolved from the MIPSPro compiler designed originally for the MIPS R10000 processor. In the process of retargeting to Intel IA64, the Pro64 code generation structure was extended so that target-dependent parts are isolated out into target-specific sub-directories. This laid the groundwork for retargeting to other processors. The process of bringing up the compiler for a new target involves mainly putting up the contents of functions located in the sub-directories created for the new target.

Pro64 uses an internal representation called CGIR in the code generation phases. CGIR is a low-level machine operation representation designed so that there is a one-to-one mapping between a CGIR operation and an instruction in the target machine. As such, CGIR is target-dependent, and its exact format is defined through the retargeting process. All the optimization phases in the code generator perform their work by operating on CGIR. The one-to-one mapping between CGIR operations and target machine instructions ensures that any improvement in CGIR will be reflected in the generated code.

To implement a compiler that generates code for a variable instruction set, one logical choice is to come up with a variable CGIR. However, this is difficult to achieve, because CGIR is derived from a pre-defined target description, and the compilation infrastructure cannot accommodate a dynamically changing instruction set. There is also the chicken-and-egg problem that the instruction set used has to be determined based on the outcome of various optimizations, and these optimizations have to be performed based on some instruction set.

We solve this compilation problem by coming up with an abstract operation representation for our compilation target, with the objective that multiple operations would eventually be compacted into single instructions in the process of arriving at the templates to enter into the dictionary. Each execution unit in the RCU inputs two operands and computes one result. Accordingly, our abstract operations are all of the format such that they take two register operands and write one register result. We also provide some operations that are not real operations in the RCU, which perform loads and stores to and from memory, and load immediate values. We refer to these operations as subsumable operations, because they could be subsumed into the instruction using special addressing modes, including immediate addressing.[5] (In the example shown later in Figure 4, op1, op2 and op7 are subsumable operations.) Loads and stores are designated as either multiport or singleport according to the areas of memory accessed. The input program is thus translated into an expanded sequence of these abstract operations. The code generator performs peephole optimizations, control flow optimizations, loop unrolling, local instruction scheduling and register allocation based on this abstract operation representation. After performing these optimizations, we introduce a post-code-generation phase, called Instruction and Dictionary Finalization (IDF), that performs the dual functions of compiling the dictionary contents and the instructions to be generated for the program code.

The use of the abstract operation representation does not preclude the compiler from using any instruction format in its final output. On the other hand, it does not pre-dispose the compiler towards any specific format either. Its use enables the code generator to optimize program code towards the core architecture of the RCU. The responsibility to produce the real program output that recognizes the various constraints in the RCU is deferred to the IDF phase, which we'll describe in the next section. Figure 3 shows the compilation scheme in the Cognigine C Compiler.

5. IDF Phase

The Instruction and Dictionary Finalization (IDF) phase is the final phase of the compiler. The input of IDF is a sequence of basic blocks generated from previous compiler stages, and the output of IDF is the assembly program that specifies the content of the dictionary and the optimized instruction sequence referring to the dictionary entries. IDF works on its input on a per basic block basis. Figure 4 gives an example to illustrate the processing performed by IDF. We'll refer to this example as we describe IDF's operations. The discussion of this section will focus on the goal of maximizing the performance of the output code. In the next section, we'll discuss how we trade-off program performance in order to accommodate the fixed dictionary size.

[5]If they are not subsumed, *move* operations will be used for them.

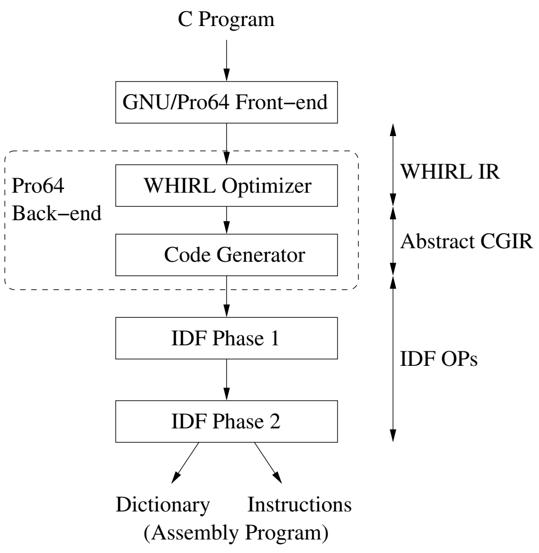


Figure 3. Structure of the Cognigine C Compiler

5.1. The IDF Approach

In our implementation, IDF is broken up into two separate phases. In the *scheduling* phase, we model IDF as a scheduling problem, in which we identify the multiple operations that can be performed in each cycle. The second phase is the *emission* phase, which scans the result of the scheduling phase on a cycle by cycle basis. Based on the operations being performed in each cycle, the emission phase generates the dictionary entry that specifies those operations, and the instruction that holds the operands for the operations performed in that cycle. The dictionary entries and the instructions together constitute the assembly output of the compiler. The emission phase is straightforward, because it just performs mechanical translation based on the result of the scheduling phase. For the rest of this section, we'll focus our discussion on the scheduling phase.

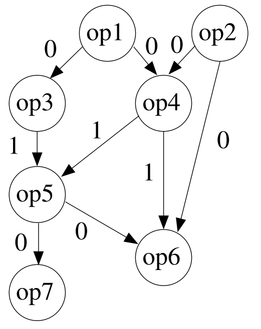
In instruction scheduling, instructions are re-ordered to minimize the overall execution time while adhering to the data dependency and resource constraints. The code of a basic block is represented as a directed acyclic graph (DAG) [11]. In a DAG $G = (N, E)$, the set of nodes N represents the set of operations and the set of edges E represents the set of dependencies among the operations. Each edge (j, k) of the DAG is labeled with a latency denoted as $l_{j,k}$. *Data dependency constraints* state that for any two operations j and k such that k has data dependency on j, k cannot be scheduled unless j is scheduled $l_{j,k}$ cycles earlier. *Resource constraints* state that for any given clock cycle i, the hardware must be able to provide sufficient resource for operations which are scheduled at cycle i. The DAG for the

Abstract CGIR:

```
op1:   r1 = load 4(p2)
op2:   r2 = move 5
op3:   r4 = and r1, r3
op4:   r5 = rshift r1, r2
op5:   r4 = add r4, r5
op6:   r5 = xor r2, r5
op7:   store 0(p3), r4
```

assuming r1, r2 and r4 have
no more use afterwards.

(a) Input program



(b) Program DAG

insn1: op1 op2 op3 op4 (op1 subsumed)

insn2: op5 op6 op7 (op7 subsumed)

(c) IDF scheduling output

dict1:	and trans0 = port0 port2	move trans1 = imm	rshift trans2 = port0 imm
dict2:	add port1 = trans0 trans2	xor port3 = trans1 trans2	

insn1: dict1 4(p2),5,r3

insn2: dict2 0(p3), r5

operands mapping:

4(p2) →	port0
5 →	imm
r3 →	port2
r4 →	trans0
r2 →	trans1
r5 →	trans2
0(p3) →	port1
r5 →	port3

(d) Generated dictionary entries and instructions

Figure 4. Example to illustrate IDF's processing

program example of Figure 4(a) is shown in 4(b).

However, the resemblance of IDF to instruction scheduling stops here. First, IDF also has to identify operations that can be folded into instructions via special instruction formats. Second, in packing multiple operations into the same cycle, IDF needs to find ways to adhere to the irregular encoding, operand and port constraints imposed by the hardware implementation to make the parallelism possible (see Section 3.3). These irregular constraints cannot be modeled as resources, and can only be taken into account based on the operations being scheduled into the same cycle. Third, IDF needs to make efficient use of the transients as scratch registers. Since a transient's value is overwritten when the same execution unit performs another operation, traditional scheduling algorithms cannot easily be made to use transients effectively.

We found that the IDF scheduling problem is best solved using an enumeration approach to instruction scheduling. There have been prior efforts to solve the problem of finding the optimal schedule for a basic block. Generally speaking, determining an optimal schedule of a basic block is known to be NP-Complete [5]. Wilken *et al.* [17] used an integer programming formulation combined with numerous techniques to produce optimal instruction schedules in reasonable time. Davidson *et al.* [7], Jonsson and Shin [9] and Narasimhan and Ramanujam [12] used branch-and-bound algorithms to find optimal solutions to resource-constrained scheduling problems. Abraham *et al.* [4] showed that implementing backtracking in a scheduler allows them to address certain processor features more effectively. In our case, finding the optimal schedule is not the main motivation for using enumeration, though near-optimal code sequences are highly desirable because the accompanying smaller memory footprints allow fitting more functionalities into the embedded processor. Instead, our main motivation is based on the fact that enumeration allows us to address a complicated machine model comprehensively and effectively. We also found that we can turn various irregular machine constraints to our advantage by using them to prune the search space, thus speeding up the enumeration.

5.2. IDF Scheduling Algorithm

Given a program in the form of a DAG, IDF tries to find the best schedule of the operations. For a schedule that is being formed, the scheduling process determines the next instruction to add to the schedule. We define $x_{op,c}$ such that $x_{op,c} = 1$ if operation *op* is scheduled at cycle c, and $x_{op,c} = 0$ otherwise. The scheduling problem is to construct a minimum length schedule, which assigns a value (0 or 1) to each variable while satisfying all the constraints.

We can model the scheduling solution space in the form of a binary decision tree called enumeration tree [13, 18, 19], as illustrated in Figure 5. Each node of the

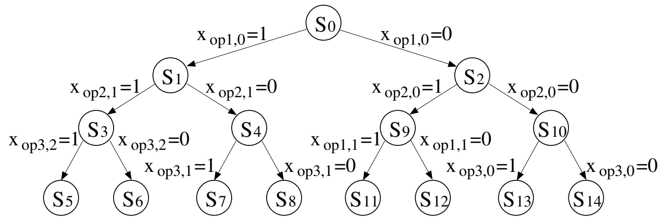


Figure 5. An enumeration tree.

tree represents partial schedules at various stages of formation. At the root of the tree, no decision has been made, and the schedule is empty. As each edge is traversed, the scheduling decision represented by the edge label is made, and is applied to all its successor tree nodes. This two-way branching scheme was originally used by Darkin [6].

The key to improving the performance of such a search algorithm is to reduce the number of valid schedules that are actually examined, in effect pruning the search space. We use an absolute lower bound of the schedule length, computed by analyzing the resources consumed on the critical path via *relaxed scheduling*[14, 10], as our initial goal, denoted as $bound_{sch}$. We prune off a search path at the first indication that the shortest schedule that can be attained in the current path cannot be shorter than or equal to $bound_{sch}$.

The IDF Scheduling Algorithm is given in Figures 6 and 7, in which function *IDF-Scheduling()* drives the enumeration process and function *Schedule-Op()* traverses each edge in the search tree. To schedule an input DAG, IDF uses s to maintain the partial schedule being formed. s is empty at the beginning, and operations will be inserted into s one by one in increasing cycle number as they are scheduled. To model the resource constraints, IDF uses a reservation table to monitor the hardware resources which are occupied by the scheduled operations at each cycle. $bound_{sch}$ represents the current goal of the schedule length; the algorithm will halt and return the final schedule s, if all the operations are scheduled and s is not longer than $bound_{sch}$. If the scheduling is unsuccessful, $bound_{sch}$ is lengthened by 1 cycle and the scheduling is restarted.

An operation is said to be ready if (1) this operation has not been scheduled yet; (2) all the data dependency constraints imposed on this operation are satisfied; and (3) the hardware resources consumed by this operation are available at this cycle. For each operation *op*, we compute and update the following three parameters:

1. release time (r_{op}) — This is the earliest cycle at which *op* can be scheduled.
2. deadline (d_{op}) — This is the latest time *op* can be scheduled in order for the final schedule to be not longer than $bound_{sch}$ cycles.
3. scheduling forms ($forms_{op}$) — This lists the possible forms of scheduling *op*, which includes scheduling it as a standalone operation, or subsuming it into the instruction via special address modes. If maintained as a standalone operation, this also indicates if its result will be left only in a transient (see Section 3.2).

```
function IDF-Scheduling
    Input: program DAG $G = (N, E)$
    Output: a feasible schedule $s$
begin
    schedule $s \leftarrow \emptyset$
    initialize $bound_{sch}$
    loop forever
        for each $op \in N$
            initialize $r_{op}, d_{op}$ and $forms_{op}$
        endfor
        $op \leftarrow$ Select-Op($N$)
        for each $f \in forms_{op}$
            if Schedule-Op($G, s, op, true, f$) = $true$ then
                return $s$
            endif
        endfor
        if Schedule-Op($G, s, op, false$) = $true$ then
            return $s$
        endif
        $bound_{sch} \leftarrow bound_{sch} + 1$
    endloop
end
```

Figure 6. The IDF scheduling algorithm

Function *Select-Op()* is invoked to select an operation *op* from a set of ready operations. If more than one ready operations exist, the operation with the earliest deadline is picked. The selected *op* is always scheduled at the cycle r_{op}. After selecting *op*, *Schedule-Op()* is called to explore each method of handling it. The possibilities are: (1) scheduling it at the current cycle, in which case it in turn explores all the possible forms given by $forms_{op}$; (2) not scheduling it at the current cycle. *Schedule-Op()* is a recursive function that corresponds to descending each edge in the enumeration tree. *Schedule-Op()* returns true if it successfully completes the scheduling without exceeding $bound_{sch}$.

In function *Schedule-Op()* shown in Figure 7, if the parameter *schedule-it* is true, it will schedule *op* to the current partial schedule s at cycle r_{op}, set its form according to the parameter *form* and mark *op* as a scheduled operation. To reflect the resource consumed by *op*, the reservation table is updated. If the parameter *schedule-it* is false, *op* is not scheduled at the current cycle, and the release time r_{op} is in-

```
function Schedule-Op
    Input: $G, s, op$, schedule-it, form
    Output: $true$ if $s$ is feasible; otherwise, $false$
begin
    if schedule-it = $true$ then
        append op to $s$
        mark op as a scheduled operation
        $form_{op} \leftarrow form$
        $d_{op} \leftarrow r_{op}$
        update reservation table
    else
        $r_{op} \leftarrow r_{op} + 1$
    endif
    if Tighten-Bounds($G$) = $false$ then
        return $false$
    endif
    $op' \leftarrow$ Select-Op($N$)
    if $op' = null$ then
        return $true$
    endif
    if $r_{op} < r_{op'}$ then
        $insn \leftarrow$ set of operations in $s$ scheduled at $r_{op}$
        if Compile-Dict-Entry( $insn$ ) = $null$ then
            return $false$
        endif
    endif
    save current state
    for each $f \in forms_{op'}$
        if Schedule-Op( $G, s, op', true, f$) = $true$ then
            return $true$
        endif
        restore state
    endfor
    if Schedule-Op( $G, s, op', false$) = $true$ then
        return $true$
    endif
    return $false$
end
```

Figure 7. The edge traversal algorithm

cremented by one. Given the scheduling (or unscheduling) decision that has been made, there exists the opportunity to tighten the release times and deadlines for the upcoming unscheduled operations. Function *Tighten-Bounds()* is called to perform this function. *Tighten-Bounds()* will return $true$ if there is no operation whose release time is later than its deadline; otherwise, it will return $false$, because this indicates that no feasible schedule exists. Bounds tightening allows us to detect and abandon infeasible search paths early, thus speeding up the enumeration process. In performing bounds tightening, we take into account the various limitations imposed by the architecture (see Section 3.3).[6]

At this point, function *Select-Op()* invoked as before will choose another ready operation, say op', to schedule next. If no more operation is available, the complete schedule s has been successfully formed, and *Schedule-Op()* returns $true$. If op' exists and $r_{op'} > r_{op}$, it means we have determined all the operations to be performed at cycle r_{op}. As a result, all the operations at cycle r_{op} will form an instruction $insn$. *Compile-Dict-Entry()* is called to compile the dictionary entry for $insn$. The details of *Compile-Dict-Entry()* will be described in Section 6.2. If *Compile-Dict-Entry()* is successful in compiling $insn$, *Schedule-Op()* will move on to schedule op'. This again involves exploring the different methods of handling it, as in function *IDF-Scheduling()*. The difference here is that it has to save the state of the scheduling tables and parameters so it can restore the state each time it backs up to take another path in the search.

Figure 4(c) shows the result of the IDF scheduling phase with our program example. The seven abstract operations are scheduled into the two instructions $insn1$ and $insn2$. The second emission phase of IDF will compile these two instructions into the two dictionary entries and two variable instructions shown in Figure 4(d).

6. Dictionary Compilation

In this section, we describe the techniques we use to address the fixed dictionary size in the VISC architecture. The dictionary can be viewed as performing two different roles in program execution: the *functionality* role and the *performance* role. The functionality role refers to the fact that it defines the different types of operations that can be performed during execution. The performance role comes from the fact that operations can be performed in parallel only if they are so specified in the dictionary. Because large programs are likely to exhibit a greater variety of operations to be performed, they will use up more dictionary space before performance is even considered. Likewise, an entry that specifies many operations has less probability of being

[6]It is beyond the scope of this paper to get into a detailed discussion of bounds tightening. General discussion of this topic can be found in [14, 17].

re-used, because it is hard to come up with exactly the same combination of operations. In our experience, we found that between 60 to 70 entries are enough to meet the functionality demand. On the other hand, under aggressive scheduling performed by IDF, dictionary entries can be used up very quickly. Thus, it is necessary to come up with techniques to minimize dictionary consumption while still generating efficiently scheduled code.

We implement this dictionary minimization functionality by extending the IDF framework presented in the last section. This is done by biasing the IDF scheduling algorithm towards re-using existing dictionary entries instead of creating new ones. To do this, IDF builds and maintains a table of the dictionary entries that have been generated as compilation progresses. At the beginning of compilation, the dictionary is empty, and IDF adds entries to it as it creates new ones.[7] During its scheduling, IDF performs efficient dictionary lookup as it tries to re-use previously generated entries.

6.1. Dictionary Lookup

To speed up the dictionary lookup, we build sets of existing dictionary entries based on operation names. Denoting the operation name of *op* by $opcode_{op}$, $Entries[opcode_{op}]$ then gives the set of existing dictionary entries that perform $opcode_{op}$ as one of its operations. All the dictionary entries of a set $Entries[opcode_{op}]$ are sorted in increasing operation count order, so that entries with larger operation count are placed later. Via this setup, the first dictionary entry found that satisfies all the requirements of the operations performed in an instruction will contain minimal extra operations.

Figure 8 gives the dictionary lookup algorithm. Given an instruction $insn$ formed during IDF's scheduling, the algorithm starts by quickly computing a much smaller set of entries S via:

$$S \leftarrow \bigcap_{op \in insn} Entries[opcode_{op}]$$

The rest of the algorithm searches for the best entry. If the entry contains operations not performed in $insn$, those operations must be rendered harmless by providing dummy operands to them. This implies: (1) the operation must not create any extra constraint to scheduling by its resource usage; and (2) its output must not overwrite any memory, live register or transient value.

[7] We also provide the facility whereby dictionary entries in previously compiled object files and libraries can be extracted and used in the current compilation.

```
Algorithm Dictionary-Lookup
    Input: instruction $insn$
    Output: a dictionary entry for $insn$
begin
    $S \leftarrow$ all the dictionary entries
    for each $op_i \in insn$
        $S \leftarrow S \cap Entries[opcode_{op_i}]$
    endfor
    for each dictionary entry $entry_i \in S$
        for each $op_i \in insn$
            $found \leftarrow false$
            for each unmarked $op_j \in entry_i$
                if $op_i$ matches $op_j$ then
                    $found \leftarrow true$
                    mark $op_j$
                    break
                endif
            endfor
            if not $found$ then
                break
            endif
        endfor
        if not $found$ then
            continue
        endif
        if an unmarked op in $entry_i$ cannot be dummy
            then continue
        endif
        return $entry_i$
    endfor
    return $null$
end
```

Figure 8. The dictionary lookup algorithm

6.2. Dictionary Usage

To make the best use of the fixed dictionary space, the compiler should use up the entire dictionary as it completes the compilation of the entire program. But because the compiler does not know when it is compiling the last code segment of the program, it always has to reserve some unused dictionary space for later use. Without any help from the user, the compiler can only estimate the progress of compilation by keeping track of how full the dictionary is. But program varies greatly in size. In general, it is impossible for the compiler to guarantee that it will not overflow the dictionary while it tries to maximize dictionary usage.

We provide the *dict_usage* compilation option to let the user control dictionary usage during compilation. This same option is also provided as a *pragma* so its value can be set differently at various points in the program. The value of this option ranges from 0 to 10. The default value of 10 designates maximum dictionary usage, and IDF is allowed to freely create new dictionary entries as it searches for the best schedule. The value of 0 indicates minimal dictionary usage, and IDF will avoid creating new dictionary entries at all cost. The values from 1 to 9 represent gradual gradation of the dictionary usage strategy between the two extremes. By experimenting with the *dict_usage* option, the user can control IDF to produce good code while getting the most out of the fixed dictionary space.

We incorporate dictionary usage decisions into the IDF scheduling algorithm by strengthening its search criteria. The IDF scheduling algorithm given in Figure 6 uses $bound_{sch}$ as the search goal in order to achieve low cycle counts. We impose an additional search goal called $bound_{dict}$, which represents the number of new dictionary entries allowed to be created in scheduling a basic block. The exact value of $bound_{dict}$ depends on the state of the dictionary: with more pre-existing dictionary entries, it is easier to find re-usable entries, and $bound_{dict}$ will be adjusted lower.

The additional constraints related to dictionary usage are dealt with in the function *Compile-Dict-Entry()* called in the edge traversal algorithm of Figure 7. Whenever IDF's scheduling creates a new instruction $insn$, it calls this function to check if the instruction being created satisfies the current dictionary creation criteria. If there is a pre-existing dictionary entry for $insn$, *Compile-Dict-Entry()* will always return $true$. If creating a new entry will cause $bound_{dict}$ to be exceeded, it will return $false$; otherwise, it will go ahead to create a new entry and return $true$.

We can greatly speed up IDF's enumeration under low values of *dict_usage* by adopting some strategies as to what kind of dictionary entries are allowed to be created. When *dict_usage* is 0, only single operation dictionary entries with no immediate addressing mode or transient addressing mode are allowed. Under this mode, we can guarantee that dictionary overflow will never occur regardless of program size. When *dict_usage* is 1, we start to allow immediates in instructions. At 2, we allow at most two operations per entry. At 3, we start to allow use of transients in the scheduling. At 4, we start to allow more than two operations per entry. These additional restrictions shrink the search space substantially while promoting dictionary re-use.

7. Results

The techniques we presented in this paper have been implemented in the Cognigine C Compiler. In this section, we provide measurement data to show the effectiveness of the techniques. Our measurements are focused on our implementation of the IDF phase as a standalone module in the compiler (see Figure 3) that accepts input produced by the Pro64 code generator. The output of the IDF phase is an assembly program in the form of the Cognigine Assembly Language. Since the CGN16100 is a network processor, we have selected five application programs, developed in house and written in C, that perform different network processing functions. These benchmarks are described in Table 1.

Benchmark	*Program function*
A. ProcessSOP	identifies and collects various attributes of the packet
B. ProcessIPv4	performs RFC1812 processing along with forwarding and classification lookups
C. ProcessMpls	MPLS label lookup and processing on packets
D. ProcessL2	switches the packet based on MAC address lookup
E. ProcessVlan	extracts and performs lookup of Vlan tag for a packet

Table 1. Network processor benchmarks

7.1. Performance Evaluation

Our first set of data, given in Table 2, helps us evaluate the effectiveness of IDF in generating high-performance output. The *dict_usage* option is set to 10 to direct IDF to generate the shortest possible schedules regardless of the number of dictionary entries created. In the first row, the benchmarks are referred to via the letters A through E. The second row, denoted *Operations Count*, gives the total number of abstract operations in CGIR form as output by the Pro64 code generator for each respective benchmark. The third row, $bound_{sch}$, gives the absolute lower bounds computed via relaxed scheduling, which are used as starting points in IDF's enumeration algorithm. The fourth row, *Subsumable Ops*, gives the number of operations that are subsumable out of the operations for each benchmark. The remaining rows in Table 2 gives data related to IDF's performance. The fifth row, *Subsumed Ops*, gives the number of operations actually subsumed in IDF's output. The sixth row, *Static Cycles*, gives the number of static cycles that IDF schedules each benchmark into.[8] The seventh row, *Insns*, gives the number of instructions in IDF's output excluding nop's and fixed instructions. The eighth row, *Dict Entries*, gives the number of dictionary entries required in the optimized output of each benchmark.

[8]We use static instead of dynamic cycle counts as our performance criterion because code size is of greater concern in the embedded processing space. A smaller code size enables more functionalities to be fitted into the final product.

Benchmark	A	B	C	D	E
Operations Count	803	907	916	785	758
$bound_{sch}$	537	617	620	505	483
Subsumable Ops	369	410	425	364	352
Subsumed Ops	87	88	92	81	86
Static Cycles	589	673	678	559	533
Insns	491	552	563	468	446
Dict Entries	177	215	211	182	175
Time (sec.)	151	155	150	156	147

Table 2. Compilation data at dict_usage = 10

Comparing the data for *Static Cycles* with $bound_{sch}$, it can be seen that the output of relaxed scheduling is too optimistic, because it ignores all the data dependency constraints and does not include various machine limitations in its model. Comparing *Subsumed Ops* with *Subsumable Ops* indicates that 23% of them are actually subsumed. This is not high because subsumption is less advantageous if the value appears more than once, and also because of limitations on what can fit in the instruction. Comparing the data for *Static Cycles* with *Operations Count* shows that IDF on average schedules 1.38 operations per instruction cycle. This includes nop's inserted to satisfy certain latency requirements in the hardware. This is quite respectable given that the output of relaxed scheduling yields 1.51 operations per instruction, and given that there are only two write ports in the processor.

In the data for *Dict Entries*, none of the benchmarks exceeds the current 256 entry limitation in the dictionary. On average each dictionary entry is referred to by 2.63 instructions, even though the compilation is done with *dict_usage* set at 10.

The last row of Table 2 gives the time taken by IDF in compiling each benchmark on an 800 MHz Pentium III machine. Due to the enumeration approach, the compilation time spent in the IDF phase is on the order of 100 times the compilation time spent in the Pro64 compiler. Because IDF is applied on a per-basic-block basis, the compilation time varies greatly based on the size of the basic blocks. Our benchmarks have been tuned to yield large basic blocks to minimize branch overhead. It is possible to substantially reduce the compilation time by starting with a less optimistic lower bound ($bound_{sch}$ in Table 2) so it can reach the final schedules sooner. Because the sizes of our application programs are limited by what can fit into the embedded processing chips, they are generally not large. Since we are targeting the embedded processing market, compilation efficiency is only secondary in importance compared to the quality of the generated output.

Benchmark		A	B	C	D	E
dict_usage = 10	Insns	491	552	563	468	446
	Entries	177	215	211	182	175
dict_usage = 8	Insns	497	565	570	479	453
	Entries	170	204	204	174	168
dict_usage = 3	Insns	505	576	585	490	463
	Entries	163	193	191	166	163
dict_usage = 2	Insns	543	615	623	523	500
	Entries	114	126	129	120	117
dict_usage = 0	Insns	751	856	862	735	709
	Entries	63	68	68	64	62

Table 3. Results at different dict_usage levels

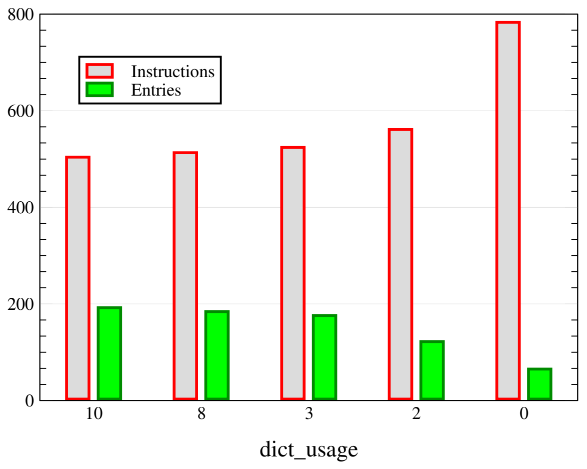


Figure 9. Instruction count vs. dictionary entries

7.2. Effects of dict_usage

In our second set of data, we look at the effects on the compilation output as we vary the *dict_usage* compilation option. We compile our five benchmark programs with *dict_usage* set at 10, 8, 3, 2 and 0. The results are given in Table 3. The rows denoted by *Insns* give the number of instructions that the programs compile to, while the rows denoted by *entries* give the number of dictionary entries created for each compilation. The data indicates that our implementation of the *dict_usage* option successfully allows the user to trade-off between dictionary consumption and program performance. In Figure 9, the averages over the five benchmarks are plotted and the trends are clearly displayed.

8. Conclusion

In this paper, we present the implementation of three innovative techniques in a production compiler to allow it to support a commercial network processor that provides a compile-time-configurable instruction set. The resulting compiler is successful in allowing the user to exploit the benefits of this architecture without resorting to assembly-level programming, which would have been painstaking and time-consuming.

The development of our compiler for Cognigine's VISC architecture was not started until after the CGN16100 chip has been designed and fabricated. The irregular parameters in the chip are hard problems for compilers to solve in general. With the completion of the compiler, it is now possible to evaluate the impacts of various architecture features on performance by feeding different hardware parameters to the compiler and analyzing the results. Such data will be used to influence the design decisions in future versions of the VISC architecture to further improve performance. There is potential for more interesting results to come in this area.

9. Acknowledgements

The authors would like to thank Kannan Narayanan for his implementation efforts in various parts of the compiler, Mike Bershteyn for his supervision and support of this project, and Gagan Gupta and Rupan Roy for insights into the VISC architecture. The application benchmarks used in this paper were contributed by Suresh Govindachar, Pavan Gupta, Sukhesh Halemane and Rick Reitmaier.

References

[1] *The ARCtangent-A4 Microprocessor Core*. ARC International.

[2] *Xtensa Overview Handbook*. Tensilica Inc.

[3] *CGN16100 Network Processor User Manual*. Cognigine Corp., 2002.

[4] S. Abraham, W. Meleis, and I. Baev. Efficient backtracking instruction schedulers. In *Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 301–308, October 2000.

[5] D. Bernstein, M. Rodeh, and I. Gertner. On the complexity of scheduling problems for parallel/pipelined machines. *IEEE Transaction on Computers*, 38(9):1308–1313, September 1989.

[6] R. Darkin. A tree search algorithm for mixed integer programming problems. *Computer Journal*, 8:250–255, 1965.

[7] S. Davidson, D. Landskov, B. Shriver, and P. Mallett. Some experiments in local microcode compaction for horizontal machines. *IEEE Transaction on Computers*, C-30(7):460–477, July 1981.

[8] G. Gao, J. Amaral, J. Dehnert, and R. Towle. The sgi pro64 compiler infrastructure. *Tutorial, International Conference on Parallel Architectures and Compilation Techniques*, October 2000.

[9] J. Jonsson and K. Shin. A parameterized branch-and-bound strategy for scheduling precedence-constraint tasks on a multiprocessor system. In *Proc. of the Int'l Conf. on Parallel Processing*, pages 158–165. IEEE, August 1997.

[10] M. Langevin and E. Cerny. A recursive technique for computing lower-bound performance of schedules. *ACM Transactions on Design Automation of Electronic Systems*, 1(4):443–456, October 1996.

[11] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[12] M. Narasimhan and J. Ramanujam. A fast approach to computing exact solutions to the resource-constrained scheduling problem. *ACM Transactions on Design Automation of Electronic Systems*, 6(4):490–500, October 2001.

[13] G. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, Inc., 1988.

[14] M. Rim and R. Jain. Lower-bound performance estimation for the high-level synthesis scheduling problem. *IEEE Transactions on Computer-Aided Design*, 13(4):451–458, April 1994.

[15] S. Talla. Adaptive explicitly parallel instruction computing. *PhD thesis, New York University*, December 2000.

[16] J. Wagner and R. Leupers. C compiler design for an industrial network processor. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES 2001)*, June 2001.

[17] K. Wilken, J. Liu, and M. Heffernan. Optimal instruction scheduling using integer programming. In *Programming Language Design and Implementation*, pages 121–133. ACM SIGPLAN, June 2000.

[18] H. Williams. *Model building in mathematical programming*. John Wiley & Sons, Inc., 1993.

[19] L. Wolesey. *Integer Programming*. John Wiley & Sons, Inc., 1998.