Bitwidth Cognizant Architecture Synthesis of Custom Hardware Accelerators

Scott Mahlke, Rajiv Ravindran, Michael Schlansker, Robert Schreiber, Timothy Sherwood
Compiler and Architecture Research Group
HP Laboratories Palo Alto

HPL-2001-209
August 30th , 2001*

E-mail: {mahlke, schlansk, schreiber}@ hpl.hp.com, rajiva@cse.iitk.ac.in, sherwood@cs.ucsd.edu

application-specific design, architecture synthesis, bitwidth, clustering, embedded system, hardware accelerator, operation scheduling, resource allocation

PICO is a system for automatically synthesizing embedded hardware accelerators from loop nests specified in the C programming language. A key issue confronted when designing such accelerators is the optimization of hardware by exploiting information that is known about the varying number of bits required to represent and process operands. In this paper, we describe the handling and exploitation of integer bitwidth in PICO. A bitwidth analysis procedure is used to determine bitwidth requirements for all integer variables and operations in a C application. Given known bitwidths for all variables, complex problems arise when determining a program schedule that specifies on which function unit and at what time each operation executes. If operations are assigned to function units with no knowledge of bitwidth, bitwidth-related cost benefit is lost when each unit is built to accommodate the widest operation assigned. By carefully placing operations of similar width on the same unit, hardware costs are decreased. This problem is addressed using a preliminary clustering of operations that is based jointly on width and implementation cost. These clusters are then honored during resource allocation and operation scheduling to create an efficient width-conscious design. Experimental results show that exploiting integer bitwidth substantially reduces the gate count of PICO-synthesized hardware accelerators across a range of applications.

* Internal Accession Date Only Approved for External Publication

This paper was presented at the 5th International Workshop on Software and Compilers for Embedded Systems, March 2001, and a version will appear in IEEE Transactions on Computer Aided Design, 2001.

1 Introduction

As the cost of complex chips decreases, the markets for PDAs, MP3 players, cellular phones, toys, games, network routers, and other specialized, high-performance electronic devices is growing explosively. Many of these devices perform computationally demanding processing of images, sound, and video or packet streams. To reduce cost and power consumption, the electronic components of these devices are now often realized as a single application-specific IC, or ASIC. In many such ASICs, specialized nonprogrammable hardware accelerators (NPAs) execute parts of the application that would run too slowly if implemented in software on an embedded programmable processor. Rapid, low-cost design, low production cost, low energy consumption, and high performance are important in these designs.

In order to reduce design time and design cost, the HP Labs *Program-In-Chip-Out* (PICO) project is focused on automating the design of NPAs from high-level specifications. Source code (in a subset of C) for a performance-critical loop nest is used as a behavioral specification of an NPA. The PICO system compiles the source code into a custom hardware design in the form of a parallel, special-purpose processor array. The system produces a VHDL design for the array, its control logic, its interface to memory, and its interface to a host processor.

PICO's goal is to synthesize hardware systems having minimal cost over a range of computational rate requirements. This paper presents analysis and optimization techniques that are needed to synthesize cost-effective hardware when function units (FUs) process program operations having differing integer precision requirements. Techniques have been developed to provide required bitwidth information on all program variables and to optimize hardware using this bitwidth information.

Our work was based on an existing PICO system that had no capability for analyzing or optimizing bitwidths. We needed an approach that provided accurate bitwidth information for use during hardware optimization. While some bitwidth information was directly available by inspecting the code (e.g. the size of a constant), other information could only be provided by the user. Thus, a facility for acquiring user-provided bitwidths was needed. Further, it was unreasonable to expect that a user decorate the bitwidth of every temporary within a program. Not only is this process tedious, many temporaries are created throughout the optimization process and the user is not even aware of their existence. Thus, we needed to develop a bitwidth analysis approach to determine the required bitwidth for all program data. In our approach, users define the bitwidth of selected variables through declarations in the source code. With knowledge of these declarations, opcode semantics, and widths of known constants, bitwidth analysis derives the required width for all program variables, expressions, and operations. The bitwidth analysis approach that is presented here is simple, efficient, and produces reasonably accurate results.

Hardware optimization using bitwidth information is a very complex problem. When each FU processes only a single program operation, the precision of each FU can be precisely tailored to the needs of this single operation. In this case, optimization is simplified to a task of hardware pruning. However, when FUs process multiple operations, the benefits of width-sensitive optimization are often diluted. When a single FU processes a mix of narrow and wide operations, it must support the widest operation that executes on it irrespective of the width of the narrowest operation. If operations are assigned to FUs with no knowledge of bitwidth, hardware is wasted as narrow and wide operations are assigned to FUs. It is therefore desirable to carefully assign operations of similar width to a common FU.

In the approach described here, all FUs use the standard C language representations for processing operands of varying width. This exploits most of the advantage available in treated examples, and is consistent with PICO's high-level synthesis heuristics and low-level synthesis capabilities. FUs are customized only in the number of bits that they process. Operands are reformatted using zero fill, sign extension and truncation. In support of this approach, we define the bitwidth of a variable to be the number of bits required to represent the variable over the range of values it can take on. If the variable is a signed integer, its bitwidth is the number of bits required in two's complement. If unsigned, then its bitwidth is the number of bits required to hold the largest attainable positive value.

The task of synthesizing hardware requires complex optimization problems to be solved.

These arise when operations of varying width are assigned to a heterogeneous set of FUs each potentially capable of executing multiple operation types (e.g. an ALU). A key goal for PICO is to provide a family of hardware solutions that vary in both cost and performance. Low performance solutions should be less expensive while high performance solutions cost more. In order to achieve this objective, processors are synthesized so as to be adequately powerful to process data at a given computation rate yet minimum in cost. If costs are to diminish as the chosen processing rate is decreased, a strategy is needed to use the same hardware unit to process more than one program operation when low processing rates are adequate. This implies that each FU may potentially processes program variables of differing width.

A scheduler chooses the FU and time at which each operation takes place. The machine cost is strongly dependent upon how well the scheduler makes these choices. To make scheduling aware of width, we employ a new technique, *width clustering*, in which operations with similar bitwidths are grouped into clusters before scheduling. Width clustering takes into account each operation's type and width and uses this information to identify width clusters that help minimize FU cost.

By binding operations of the same type to a common FU, cost is reduced as the FU is specialized (e.g. an adder as opposed to an ALU). The scheduler can channel expensive operations like divides into a common FU to avoid proliferation of expensive FUs. By binding operations of the same width to a common FU, cost is reduced as FUs that process only narrow operands are themselves narrow. When FUs process operands of similar width, the costs of the registers and switches that connect FUs are also reduced. However, difficulty arises because these criteria often compete.

Width clustering addresses these complex tradeoffs before scheduling begins as it groups operations into clusters. Width clusters guide resource allocation and scheduling to produce more a efficient design. Clusters are formed by analyzing the types and widths of all operations. Operations that may share resources to reduce cost are placed in the same width cluster. After clusters are formed, hardware resources are allocated separately for each cluster. This allocation is then used by a scheduler that uses these resources to satisfy all computational needs. During scheduling, the binding of operations is restricted to FUs from their own cluster. This produces a cost-sensitive binding of operations to resources based on operation bitwidth and type.

We believe that width clustering represents a first attempt to synthesize hardware over a range of computation rates while exploiting both type and width information for each operation. Width clustering produces efficient hardware by selecting hardware from a complex and heterogeneous library of FUs each capable of executing one or more operation types. Results indicate substantial improvements in the cost of generated hardware.

2 NPA Synthesis in the PICO System

The overall structure of PICO is shown in Figure 1. A C loop nest is identified by the spacewalker (a design space exploration tool) as the application component to be synthesized and provided to the loop parallelizer to begin the process. Both the number of processors and the computational rate for of each processor are specified by the spacewalker as input to the synthesis process. These parameters collectively determine the computational rate at which the loop nest can be processed and are specified either automatically by PICO's spacewalker or manually by a user. PICO designs a nonprogrammable processor array for the given loop nest consistent with this computational rate specification. The RTL design (in VHDL) is written to an output file. PICO also generates performance and gate count measurements for the NPA. See [1] for a full description PICO's NPA synthesis capabilities.

We now provide an overview of PICO by briefly describing each of its components.

Spacewalker: PICO's spacewalker is a complex heuristic engine that drives system synthesis. In general, multiple application components must be accelerated on one or more customized hardware processors. Processors take on more than one form including the PICO-NPAs discussed here as well as PICO-VLIWs (VLIWs customized to specific applica-

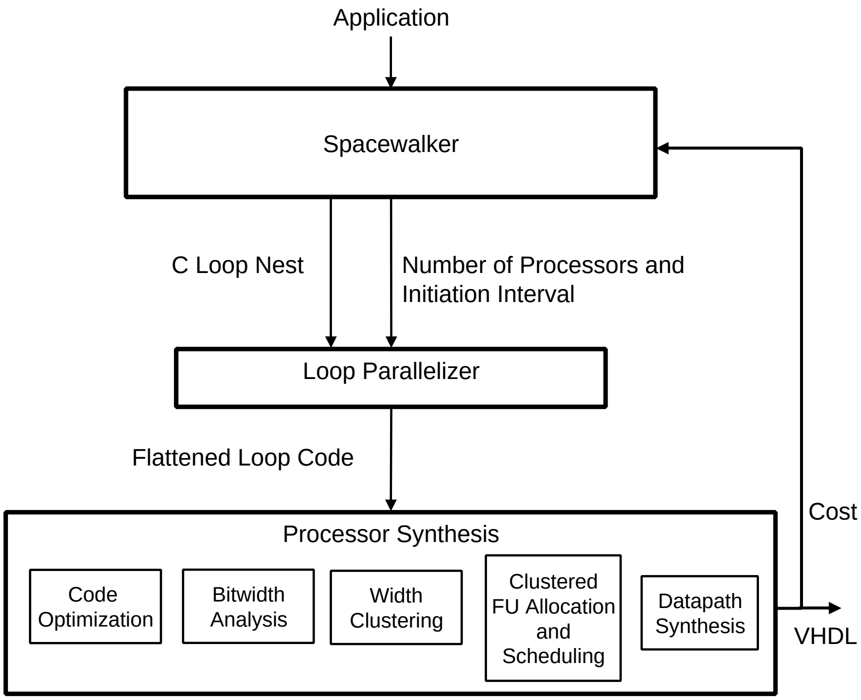


Figure 1: The PICO NPA design system.

tion requirements) [2]. A limited chip area is available for these diverse needs. Further, given a desired computation rate, the hardware cost or chip area required by a suitable accelerator is not known until the synthesis process is at least partially completed. To synthesize a complex system having optimal performance and cost, PICO's spacewalker selects candidate performance goals for specific application components and it requests that these components are synthesized to evaluate their cost. The merits of this choice can than be evaluated at the system level, and the choice can be adopted or adjusted before full system synthesis proceeds. This paper focuses exclusively on techniques for optimizing a loop nest to produce a single PICO-NPA at a candidate computation rate as requested during spacewalking.

Loop Parallelizer: The loop parallelizer is given a nest of counted loops and analyzes and exploits parallelism within that nest by generating a high-level plan called an iteration schedule. The iteration schedule determines a temporal (what time) and spatial (what processor) plan for all loop iterations. In order to maximize scheduling freedom, the loop parallelizer perfectizes the input loop nest. The resulting perfect nest of counted loops is flattened into a single loop with a trip count that is the product of the loop trip counts in the original nest. In this form, the iteration scheduler gains additional freedom in organizing the loop nest for parallel execution and parallelism is limited only by the code's essential data dependences.

A valid iteration schedule must satisfy the following properties; each processor's loop code executes a precisely specified subset of all loop iterations and every loop iteration is executed on some processor. This plan is symmetric among processors. A single loop body is generated that is executed in lock-step parallel manner on all processors. The plan has the property that all dependence constraints can be met both within each processor and among processors at the requested computation rate.

Processor Synthesis: The process of creating a customized datapath from the loop body is shown in the Processor Synthesis box in Figure 1. The goal is to achieve the requested throughput for the given code with minimum hardware cost. This is performed by first allocating a set of FUs and then software pipelining the loop code. Software pipelining generates loop schedules for PICO NPAs. Software pipelining creates a single program schedule for all iterations that can be initiated at a constant rate called the *initiation interval* (II). The software pipeliner can bind multiple operations to each FU. The FU's hardware realization will be determined after scheduling and it will be made as wide as the widest of these operations.

We now describe the modules that are used within processor synthesis.

Code Optimization: This phase is performed by Elcor, a retargetable VLIW compiler [3]. After classical optimizations, if-conversion removes any branching within the loop body. The resulting branch-free loop body is suitable for software pipelining.

Bitwidth Analysis: This phase infers the bitwidth required to represent every value computed in the loop. Our approach is presented in Section 3.

Width Clustering: The set of operations is partitioned into subsets of operations having similar width using the heuristics discussed in Section 4.

Clustered FU allocation and scheduling: Before software pipelining begins, a set of resources must be allocated that are suitable for executing the loop at the given single-processor rate. Rather than allocating a single set of resources where each resource can be used to execute any compatible operation, resources are allocated in clusters. Within each operation cluster, we allocate (by solving a small mixed integer linear program) a set of FUs that is powerful enough to perform the cluster operations at the desired II [4]. Each cluster of FUs is then characterized by a machine description for use by the Elcor software pipeliner.

Hardware is synthesized by first generating a software schedule that decides on which FU and at what time each operation occurs, and then, by more mechanically generating a datapath during datapath synthesis. Each processor in the array is heavily pipelined. The computation of a single iteration typically requires more than II cycles, so that there will be several iterations in the pipeline at any given time. The software pipeliner schedules operations so that dependences among operations are satisfied both within each iteration and for any carried dependences between iterations. Moreover, resource conflicts must be avoided: the scheduler ensures that two operations are not scheduled on the same FU at the same time. For simplicity and consistency with PICO's current low-level hardware synthesis capability, we assume that all FUs are fully pipelined and able to begin a new computation on every cycle. It follows that a given FU can be assigned at most II operations from the loop body.

Datapath Synthesis: The scheduler makes many difficult heuristic decisions regarding how resources are to be used. By finalizing these decisions, the synthesis of the registers, switches, and interconnect needed to maintain and transport operands within and among FUs within a processor becomes somewhat more mechanical. Datapath synthesis generates a customized datapath for a single processor and then replicates that datapath for all processors. Each datapath is connected where needed with sibling processors to yield an array of processors capable of executing the all iterations at the desired aggregate rate.

3 Bitwidth Analysis

Bitwidth analysis infers the bitwidth of every variable in a program segment. The analysis operates on the assembly-level internal representation in Elcor. Each reference to a register as a source or destination operand is tagged with its computed bitwidth. The results of bitwidth analysis are used by architecture synthesis to infer the sizes of the hardware components for the hardware accelerator.

PICO uses initial bounds on the bitwidth for specific variables and iterative constraint propagation to identify adequate bitwidths for all variables. Initial bounds have multiple sources. First, conventional C variable types (bool, char, and short) provide important bitwidth information. The exact widths of all constants are directly known. We also give the user more fine-grained control over bitwidths of variables: a pragma specifying an arbitrary bitwidth (e.g., 5 bits) may be optionally supplied after each variable declaration in the C source code. Values read from or written to an external location, such as memory, are ideal candidates for user bitwidth annotation. These values are not analyzable and the compiler must assume the worst case in the absence of user intervention.

Another source of initial bounds is the PICO loop parallelizer, which introduces a number of variables into the code as it transforms the original program to parallel form. Bounds on these values are generally known by the loop parallelizer, and their required widths are therefore known. However, these widths are not readily visible by direct inspection of the code after loop parallelization. The loop parallelizer inserts additional pragmas within resultant code to provide this information.

These bounds on bitwidths for all program variables provide a starting point for iterative constraint analysis. Iterative constraint analysis can refine or narrow the bitwidths for many values by repeatedly propagating width constraints through the program. The width of a variable is constrained by two factors. First, the width is limited by the amount of useful data available when the variable is defined. This is referred to as the *def constraint*. For example, 16 bits are not necessary to hold the result of adding two 3-bit numbers – 4 bits is enough. Second, a value need not retain more bits than the number needed by its uses. This is referred to as the *use constraint*. For example, a 32-bit quantity contains unneeded data if it is only used in 10-bit add operations.

The individual operations are connected via define-use and use-define chains such that every define of a variable is connected to the operations that consume that value and the reverse. We repeatedly apply the def and use constraints to get ever tighter restrictions on variable widths until we converge to a stable solution. This approach is a natural extension to standard forward and backward dataflow analysis techniques [5].

The iterative constraint propagation is best explained by breaking it down into its three constituent components: opcode transfer functions, forward analysis, and backward analysis. Each is discussed in the remainder of this section followed by an example of the entire process.

3.1 Opcode transfer functions

At the individual operation level, there is an opcode-specific calculation that determines the flow of information through the operation. For example, when two 6-bit quantities are added, it is known the result is not larger than 7 bits. Similarly, when an add has a 10-bit result, it is known the inputs need not be larger than 10 bits. Such functions, referred to as *opcode transfer functions*, are determined for every opcode in the compiler's instruction set. They are broken down into forward opcode transfer functions to specify the rules for computing output widths of an operation given its input widths, and backward opcode transfer functions to specify the rules for computing input widths of an operation given its output widths.

The opcode transfer functions for some commonly occurring integer arithmetic opcodes are presented in Table 1. The forward transfer function for add states that the destination width is the maximum of the two source widths plus one. In essence, a single carry-out bit from the larger number could be generated, hence one additional bit is required. The backward transfer function for add states that the width of both sources is equal to the width of the destination. Since an add only propagates information from the low-order bits to the high-

Table 1: Opcode transfer functions for common integer opcodes. The form of an operation is: dest = src1 opc src1. We use d, s1, and s2 to represent the widths of dest, src1, and src2, respectively.

Opcode	Forward	Backward
add	d = MAX(s1, s2) + 1	s1 = d, s2 = d
subtract	d = MAX(s1, s2) + 1	s1 = d, s2 = d
unary negate	d = s1 + 1	s1 = d
multiply	d = s1 + s2	s1 = d, s2 = d
divide	d = s1 + 1	s1 = max_width, s2 = max_width
left shift by const	d = s1 + C	s1 = d - C, s2 = max_width
right shift by const	d = s1 - C	s1 = d + C, s2 = max_width
compare	d = 1	s1 = max_width, s2 = max_width
bitwise and	d = MIN(s1, s2)	s1 = d, s2 = d
bitwise or	d = MAX(s1, s2)	s1 = d, s2 = d
bitwise xor	d = MAX(s1, s2)	s1 = d, s2 = d
bitwise complement	d = max_width	s1 = d

order bits, an m-bit result is only dependent on the low-order m bits of the inputs. For integer divide, the destination is no wider than the divisor. However, a maximal positive value for a particular bitwidth could be divided by negative one, thereby increasing the required width of the result by one in two's complement format. Conversely for divide, the destination width places no constraints on the source widths. Hence, the only conclusion is that the sources are unconstrained, represented as *max_width* in the table. The table presents the transfer functions for a variety of other opcodes that are derived through similar analyses.

3.2 Forward analysis

Forward analysis repeatedly applies the def constraint to limit the output widths of all operations. Information is propagated from operation inputs to their outputs via the forward opcode transfer functions. The forward propagation phase is applied iteratively across all operations in the program until a fixed point is reached.

The algorithm for forward analysis is presented in Figure 2. The algorithm maintains two sets of widths for all of the register references in the program segment (or region) being analyzed: CW and FW. The current width or CW is the last set of stable widths that were computed. Initially, CW is determined from the variable declaration information received from the PICO frontend. The forward width or FW is the set of working widths that are computed during forward analysis. FW is initialized differently for each type of operand. For source operands that are defined externally (a live-in register, memory location, or literal), the CW value is used as the initial value. These operands are never computed in the code, thus forward analysis cannot make any conclusions about the widths of these operands. For all other source and destination operands, FW is set to uncomputed, represented as 0 in the algorithm.

The middle portion of the algorithm in Figure 2 shows the iterative forward analysis process. The source widths (FW[o,s]) for an operation are calculated by determining the widest definition of the source to reach the operation under consideration. Note that minimum between the widest reaching definition and the CW is always taken, so that the width is never increased beyond its last stable constraint. The destination widths are then computed by applying the forward opcode transfer function. The process continues until a fixed point is reached. When the fixed point is achieved, the FW widths represent the next stable and more constrained set of widths. Hence, CW is updated with FW where there are differences.

3.3 Backward analysis

Backward analysis is analogous to forward analysis with the direction of all constraint propagation reversed. The use constraint is repeatedly applied to limit the input widths of each operation given constraints on the output widths. Information is propagated from an operation outputs to its inputs using the backward opcode transfer functions.

The algorithm for backward analysis is presented in Figure 3. It is very similar in structure to the forward analysis algorithm, thus only a few differences are pointed out here. The backward width or BW is the set of working widths that are computed during backward

Compute width information for all operations in a region in the forward direction. Widths are maintained for each variable reference, operation × operand. CW contains the stable reference widths for the region. FW contains the working reference widths computed during forward analysis.

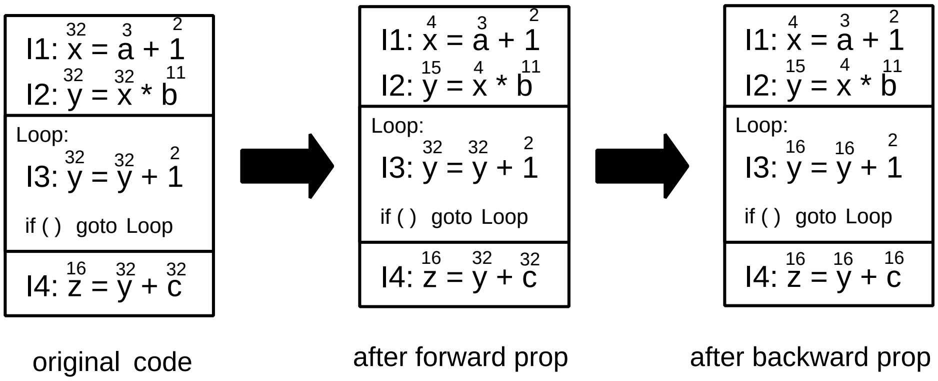
```
Procedure forward_propagation(region, CW)
  // Initialize the working reference widths, FW
  for each operation o in region in sequential order
    for each source operand s of o
      if (s $\epsilon$ (livein $\cup$ memory $\cup$ literal)) then
        FW[o,s] = CW[o,s] ;
      else
        FW[o,s] = 0 ;
      endif
    endfor
    for each destination operand d of o
      FW[o,d] = 0 ;
    endfor
  endfor
  // Iterate until reach a fixed point solution
  change = true ;
  while (change)
    change = false ;
    for each operation o in region in sequential order
      for each source operand s of o
        // Each source width is the max of its reaching defs
        FW[o,s] = MIN(MAX(FW[o,rdef[o,s]]), CW[o,s]) ;
        if (FW[o,s] changed) then
          change = true ;
        endif
      endfor
      for each destination operand d of o
        // Propagate RHS to LHS using the forward tf
        FW[o,d] = MIN(op_forward(o,d,FW), CW[o,d]) ;
        if (FW[o,d] changed) then
          change = true ;
        endif
      endfor
    endfor
  endwhile
  // Update region reference widths
  change = false ;
  for each operation-operand pair i, j
    if (CW[i,j] $\neq$ FW[i,j]) then
      CW[i,j] = FW[i,j] ;
      change = true ;
    endif
  endfor
  return change ;
```

Figure 2: Iterative algorithm for forward bitwidth analysis.

Compute width information for all operations in a region in the backward direction. Widths are maintained for each variable reference, operation × operand. CW contains the stable reference widths for the region. BW contains the working reference widths computed during backward analysis.

```
Procedure backward_propagation(region, CW)
// Initialize the working reference widths, BW
for each operation o in region in reverse sequential order
  for each destination operand d of o
    if (d $\epsilon$ (liveout $\cup$ store)) then
      BW[o,d] = CW[o,d] ;
    else
      BW[o,d] = 0 ;
    endif
  endfor
  for each source operand s of o
    BW[o,s] = 0 ;
  endfor
endfor
// Iterate until reach a fixed point solution
change = true ;
while (change)
  change = false ;
  for each operation o in region in reverse sequential order
    for each destination operand d of o
      // Each dest width is the max of its reaching uses
      BW[o,d] = MIN(MAX(BW[o,ruse[o,d]]), CW[o,d]);
      if (BW[o,d] changed) then
        change = true ;
      endif
    endfor
    for each source operand s of o
      // Propagate LHS to RHS using the backward tf
      BW[o,s] = MIN(op_backward(o,s,BW), CW[o,s]) ;
      if (BW[o,s] changed) then
        change = true ;
      endif
    endfor
  endfor
endwhile
// Update region reference widths
change = false ;
for each operation-operand pair i, j
  if (CW[i,j] $\neq$ BW[i,j])
    CW[i,j] = BW[i,j] ;
    change = true ;
  endif
endfor
return change ;
```

Figure 3: Iterative algorithm for backward bitwidth analysis.



I1: $\overset{32}{x} = \overset{3}{a} + \overset{2}{1}$
I2: $\overset{32}{y} = \overset{32}{x} * \overset{11}{b}$
Loop:
I3: $\overset{32}{y} = \overset{32}{y} + \overset{2}{1}$
if ( ) goto Loop
I4: $\overset{16}{z} = \overset{32}{y} + \overset{32}{c}$
original code
I1: $\overset{4}{x} = \overset{3}{a} + \overset{2}{1}$
I2: $\overset{15}{y} = \overset{4}{x} * \overset{11}{b}$
Loop:
I3: $\overset{32}{y} = \overset{32}{y} + \overset{2}{1}$
if ( ) goto Loop
I4: $\overset{16}{z} = \overset{32}{y} + \overset{32}{c}$
after forward prop
I1: $\overset{4}{x} = \overset{3}{a} + \overset{2}{1}$
I2: $\overset{15}{y} = \overset{4}{x} * \overset{11}{b}$
Loop:
I3: $\overset{16}{y} = \overset{16}{y} + \overset{2}{1}$
if ( ) goto Loop
I4: $\overset{16}{z} = \overset{16}{y} + \overset{16}{c}$
after backward prop

Figure 4: Example application of bitwidth analysis.

analysis. The initialization process for BW sets the width of destination operands that are either memory locations or live-out registers to CW. These operands have no consumers, thus backward analysis cannot derive any information about their widths. All other operands have their BW set to uncalculated or 0. The backward analysis is iteratively applied across all operations until a fixed point is reached.

3.4 Example

To illustrate the application of bitwidth analysis, consider the example in Figure 4. The original code consists of four instructions, two sequential instructions, a third within a loop, and a fourth after the loop. For this example, the trip count of the loop is unknown. The initial widths provided by the user are annotated above each variable in the original code. Forward propagation applies the def constraint to propagate right-hand side constraints to the left-hand side for each instruction. For I1, the addition of a 3-bit and a 2-bit quantity produces at most a 4-bit result, hence the width of x is 4 bits. For I2, the 4-bit value for x is propagated downward from I1. Then, the forward opcode transfer function for multiplication states that n bits multiplied by m bits yields at most $n + m$ bits (see Table 1). Thus, the width of y is calculated as 15 bits. Similar propagation is applied to the other instructions. Since I3 is within a loop, the forward propagation iterates until reaching a fixed point in which y is 32 bits. This result is best possible since the loop iterates an unknown number of times.

Backward analysis is applied next. The constraint of the final output, z, being no more than 16 bits is propagated. This affects the width of y and c in I4 and I3 because the 16-bit output requires only 16-bit inputs. Note that the with of c is reduced even though it is a live-in variable. I1 and I2 are not affected by the backward propagation because they already contain stronger width constraints.

In this example, a second iteration of forward analysis after backward analysis completes yields no further improvement. In fact, we have tried and failed to find a case in which applying an outer loop to the analysis process is useful. We suspect that forward iteration to convergence followed by backward to convergence achieves the best solution with this approach.

4 Width-Sensitive Architecture Synthesis

The architecture synthesis process makes all decisions needed to define efficient hardware for a given input loop nest. In this section, we describe techniques that we have incorporated into architecture synthesis that allow it to use bitwidth information to further improve the efficiency of the generated hardware.

The algorithms for architecture synthesis presented here are based on heuristics that divide a very complex problem into multiple simpler problems that are solved in phased sequence. A truly optimal strategy jointly makes all design decisions in an environment where it can establish that the selected decisions are superior or equal to any other design choice. Optimal search algorithms typically require the traversal of a combinatorial search space.

The use of bitwidth information is one example of the ongoing incorporation of additional design complexity into PICO's architecture synthesis approach. This complexity is exhibited in a number of ways. Bitwidth information adds complexity to PICO's architecture synthesis input. If this new information is to be exploited, algorithms will have to be upgraded to take this information into account. The optimization criteria used during architecture synthesis grow more complex as PICO's architecture synthesis process tries to more faithfully model variable width hardware cost. And finally, the incorporation of variable width hardware greatly adds to the diversity of designs that must be considered. Multiple design choices that were co-equal when widths were fixed now represent distinct potentially optimal choices.

A number of papers have presented cleverly contrived integer linear programming formulations of the special-purpose hardware synthesis problem [6, 7]. These formulations represent the necessary design decisions by using a large number of 0/1 integer variables. These efforts have implicit architectural limitations that constrain the search space as defined by the formulation so that solution is tractable. No doubt, they can be extended in architectural scope and to accommodate bitwidth information. The concern with these methods is the runtime of the solver: as the complexity of the available choices for architecture synthesis and the optimization criteria continue to increase, optimal search algorithms experience exponentially growing runtimes and in practice they are unacceptably slow.

Prior to the incorporation of bitwidth analysis, PICO had a heuristic, two-phase strategy for architecture synthesis. In this strategy, a first phase identifies a set of FUs of smallest cost that is capable of executing the loop body at the requisite computation rate. This minimization implicitly assumes that interconnect cost is less important than FU cost. The second phase schedules all operations on a specific FU and at a specific moment in time, thus completely specifying the higher-level architecture. The FU cost minimization uses an integer linear programming formulation that is practical due to the simplicity of the search space (there are no 0/1 integer variables) and the empirical observation that optimal solutions can be found quickly [1]. The scheduling pass is heuristic due to the very large search space and the lack of efficient and provably optimal decision making criteria.

The need to jointly consider operation width and operation type affects both the composition of the FUs in the synthesized processor as well as the detailed binding of each operation to one of these FUs. The software pipeliner is responsible for solving a difficult combinatorial search problem and uses heuristics to identify an operation schedule that meets resource and dependence constraints. Rather than adding complexity to the scheduler, we have developed a clustering phase that is invoked before scheduling. It is designed to restrict the scheduler in such a way that efficient width-sensitive designs are produced.

In general, clustering is a process of partitioning the set of operations and the set of FUs into subsets before scheduling, and constraining the scheduler to bind operations to FUs of the same cluster. Operation clustering has traditionally addressed the problem of compiling programs for predefined hardware clusters of FUs and register files [8].

PICO balances the competing costs of supporting operation width and operation type by width clustering. In width clustering, the set of operations is first partitioned into subsets having similar type or similar width. After operation clusters are formed, FUs are allocated separately for each cluster. Width clustering promotes the use of narrow FUs for narrow operations and it also channels expensive operations into a single cluster to avoid proliferation of expensive FUs.

Width clustering consists of the following three steps: 1) virtual FU assignment, 2) virtual FU clustering, 3) creation of clustered machine description. Each is discussed in the remainder of this section. An example then follows.

4.1 Virtual FU assignment

Virtual FU (VFU) assignment is a preliminary binding of operations to FUs that is directed by the cost of implementing the operations, with known width, on heterogeneous FUs. It is derived without using any data dependence information. VFU assignment provides a sample binding from which further clustering decisions are made. It does not constrain the actual bindings that are finally made. Pseudo-code for the VFU assignment algorithm is provided in Figure 5, and pseudo-code for it's supporting procedures is provided in Figure 6.

The VFU assignment has a number of inputs. A set of operations that must be implemented along with the requisite II are provided. Each operation, op, has a width, op.width, that is determined using bitwidth analysis. PICO uses a library of FUs each with a specific opcode

Assign ops in region to vfus. Returns the list of accrued vfus, vfulist. The heuristic method is utilized until the overcost of the candidate goes above a threshold (MAX_OVERCOST). Once this occurs a recursive method is used to perform the remainder of the vfu assignment.

```
Procedure vfu_assign(region, II)
  // Build oplist, a list of ops sorted from highest to lowest
  // cost of the cheapest FU to implement that operation
  oplist = build_oplist(region) ;
  while (oplist not empty)
    cvfulist = build_cvfulist(oplist, II) ;
    best_overcost = infinity ;
    for each candidate virtual function unit cvfu in cvfulist
      overcost = compute_overcost(cvfu, II) ;
      if (overcost < best_overcost) then
        best_overcost = overcost ;
        best_cvfu = cvfu ;
      endif
    endfor
    // If best_cvfu has an acceptable overcost, then keep it
    if (best_overcost ≤ MAX_OVERCOST) then
      vfulist.add(best_cvfu) ;
      oplist = oplist - best_cvfu.bound_ops ;
    // Else, use recursive approach to bind remaining ops
    else
      find_vfus_recursively(oplist, II, selected_fus) ;
      vfulist = vfulist + selected_fus ;
      oplist.clear() ;
    endif
  endwhile
  return vfulist ;
```

Figure 5: Algorithm for virtual FU assignment.

repertoire and a cost that varies with the FU's width. The VFU assignment procedure uses a cost function, fu.cost(width), that is defined for each FU. The cost depends on both the function to be implemented as well as the width of the FU implementation. These costs are calibrated from an existing standard cell library that can generate actual FUs of appropriate width and repertoire.

VFU assignment begins in the procedure vfu_assign in Figure 5. The input parameter region is an object that holds all required information about the region of the input program containing the loop nest for which hardware is to be synthesized. The parameter II specifies the initiation interval for the desired schedule. As we discuss later, two distinct heuristics

Create a list of cvfus compatible with the first operation in oplist, bind up to II operations to each cvfu.

```
Procedure build_cvfulist(oplist, II)
seed = oplist.head ;
cvfulist = create_compatible_cvfus(seed) ;
for each candidate virtual function unit cvfu in cvfulist
  num_assigned_ops = 0;
  for each operation x in oplist
    if (cvfu.fu.implements(x)  then
      cvfu.bind(x) ;
      cvfu.width = MAX(cvfu.width, x.width) ;
      num_assigned_ops = num_assigned_ops + 1 ;
    endif
    if (num_assigned_ops == II)  then
      break;
    endif
  endfor
endfor
return cvfulist ;
```

Returns the cost for the best assignment of the remaining operations to vfus. The parameter selected_fus is the 2nd return value, a list of the vfus coresponding to the returned cost.

```
Procedure find_vfus_recursively(oplist, II, selected_fus)
if (oplist is empty)  then      // Terminate recursion
  return 0 ;
endif
// For each cvfu, recursively assign remaining ops to vfus
cvfulist = build_cvfulist(oplist, II) ;
best_cost = infinity ;
best_cvfus = 0 ;
for each candidate virtual function unit cvfu in cvfulist
  unbound_oplist = oplist - cvfu.bound_ops ;
  cost = find_vfus_recursively(unbound_oplist, II,
                      selected_fus) ;
  total_cost = cost + cvfu.fu.cost(cvfu.width) ;
  if (total_cost < best_cost)  then
    best_cost = total_cost ;
    best_cvfus = selected_fus + cvfu ;
  endif
endfor
selected_fus = best_cvfus ;
return best_cost ;
```

Figure 6: Support functions for virtual FU assignment.

are embodied within the pseudo-code. A more accurate heuristic uses a recursive descent to calculate cost, while a faster heuristic terminates this descent and sacrifices the optimality of selected VFUs while accelerating the VFU assignment process.

The first action performed within vfu_assign is to invoke the function build_oplist (implementation not shown) in order to build a sorted list of operations from input code. For each operation, a cheapest FU is identified and used to determine the operation's inherent cost. An operation's, op's, cheapest FU, CFU(op), is the least expensive FU among those capable of executing op. The determination of the cheapest FU takes the operation's width into account. An operation's width is defined as the maximum width of all of its operands. This represents a limitation of current work as some operations like loads can have address width that is unrelated to the data width and the use of this maximal width is imprecise. For each FU, fu, that is capable of executing the operation, fu's cost is measured at the width needed by the operation, fu.cost(op.width). A cheapest FU is any of the FUs that minimizes this cost. After inherent costs are calculated, operations are sorted from highest to lowest cost and returned as oplist from the function call to build_oplist.

At each step in the VFU assignment procedure, a seed operation is selected from which a VFU is grown. This process begins with the call to build_cvfulist whose implementation is shown in Figure 6 (top). The function build_cvfulist identifies a seed operation as the costliest operation that has not already been bound to a VFU. Given a seed operation, a candidate VFU (CVFU) is grown for every hardware FU in the library that implements the seed. The invocation of create_compatible_cvfus(seed) creates a list containing a CVFU for each FU that implements the seed operation. Each of these CVFUs is initialized with the property cvfu.fu which identifies the FU that led to its creation.

A loop then separately processes each CVFU. A CVFU acquires additional operation bindings as an inner loop traverses the list of unbound operations from highest to lowest inherent cost. As each operation is considered, the operation is bound to the CVFU if the FU corresponding to the CVFU implements the operation and the CVFU does not already have II operations bound to it. Initially, zero operations are bound to the CVFU and the first operation processed is the seed. The seed is always compatible with and is always bound to the CVFU. Operation binding continues until the CVFU has II bound operations or the prioritized list of unbound operations is exhausted. When build_cvfulist is complete, a list of CVFUs is returned. Each CVFU has a set of operations (cvfu.bound_ops) that has been bound to it, a width (cvfu.width) corresponding to the width of the widest operation, and a hardware implementation cost for the CVFU (cvfu.fu.cost(cvfu.width)).

The algorithm uses one of two methods to determine the CVFU that is selected as the final VFU for the seed. A rapid heuristic minimizes an *overcost* function that computes the amount that the actual implementation exceeds a lower bound on the minimum possible cost. At each step in the algorithm, the VFU for the seed is selected as the minimal overcost CVFU. The overcost function is defined as

$$\text{overcost} \;=\; \text{cvfu.fu.cost(cvfu.width)} - \sum_{\text{op} \in \text{cvfu.bound\_ops}} \frac{CFU(\text{op}).\text{cost}(\text{op.width})}{II} \qquad (1)$$

The overcost measures how close the actual cost of the hardware implementation for a CVFU is to the sum of the inherent costs for all operations assigned to that FU. The CVFU having the lowest overcost is chosen as the VFU. After a VFU is identified, the process continues by selecting the next seed and growing a new VFU until all operations have been bound.

A threshold test (overcost $\leq$ MAX_OVERCOST) determines whether the rapid heuristic is acceptable or more accurate heuristics should be employed. When the overcost is unacceptably high, a fully-recursive technique is employed by calling find_vfus_recursively. The function find_vfus_recursively calls build_cvfulist to construct a list of CVFUs. For each CVFU, the total cost is calculated as the actual cost of the CVFU plus the cost of implementing all remaining operations not bound to the CVFU. This remaining cost is calculated by recursively calling find_fus_recursively with an oplist consisting of the remaining unbound operations. A minimal cost is selected over all CVFUs and returned.

The VFU assignment pseudo-code integrates a rapid heuristic and an exponential heuristic into a common algorithm. This algorithm has been used to enhance our understanding of both heuristics and to gain a better understanding of how we might wish to implement future width-clustering heuristics. We have shown on a number of examples that the fully-recursive heuristic improves on results achieved by the rapid heuristic. By setting the MAX_OVERCOST to infinity, the rapid heuristic is always used. By setting MAX_OVERCOST to a negative number, the fully-recursive heuristics always used. This allows the comparison of results derived by exclusive use of either heuristic. The fully-recursive heuristic is exponential in nature and cannot be used in a production setting for large-scale problems. However, timeout based schemes or other computation limiting schemes can be used to integrate limited recursion into the VFU assignment algorithm.

After VFU assignment is complete, a set of VFUs is defined. Every operation is bound to one VFU, with a maximum of II operations bound to a single VFU. The VFU selection and assignment heuristic of this section has chosen a set of VFUs of approximately minimal total cost. The set of VFUs and the binding of operations to them is next used to drive downstream clustering that is cognizant of the effects of both width and repertoire on FU cost.

4.2 Form operation clusters through virtual FU clustering

The purpose of this step is to partition the set of operations into operation clusters. To that end, VFU clustering is used to group VFUs based on width. The width of each VFU is determined by the widest operation assigned to that VFU. The VFUs are sorted from highest to lowest in width. A cluster is initialized when the widest unbound VFU is added to it. The width of this VFU defines the cluster width. The ratio of the cluster width to each of the remaining unbound VFUs is calculated. VFUs are added to the cluster until this ratio falls below some threshold (*e.g.,* 1.5). When the cluster is complete, the widest unbound VFU is again selected as a seed to form a new cluster. The process repeats until all VFUs are assigned to clusters. Finally, each VFU cluster gives rise to an operation cluster. All operations bound to a common VFU cluster reside within a common operation cluster.

After operation clusters are formed, the VFUs have no further use and are discarded.

4.3 Machine description creation and scheduling

Creation of the clustered machine description completes the width-clustering process. For each operation cluster, a set of FUs that can execute all operations within the cluster at the required rate is selected using integer linear programming. The integer linear program allocates FUs from a library of FUs having known cost functions. These functions relate FU width to FU cost. In the PICO library, the unit of cost is estimated gate count.

The width of each cluster is determined by the widest operation within the cluster. For each of the operations within the cluster, all FUs that implement the operation are added to that cluster's FU library. The cost for each of these FUs is evaluated at the cluster width. The integer linear program is then applied separately, for each cluster, in order to determine the initial set of FUs for the cluster.

The selection of FUs is translated into a machine description needed by the software pipeliner. A machine description for all clusters is assembled by instantiating scheduling alternatives for all allocated FUs within all clusters. For each alternative, its FU type is used to identify a machine description for the FU that is used to construct the machine description for the alternative (i.e. the instance of the FU). The software pipeliner has been altered so that it limits the binding of each operation to scheduling alternatives corresponding to FUs that are within the operation's width cluster. After the machine description is constructed, the software pipeliner is then used to determine a FU and time for all operations.

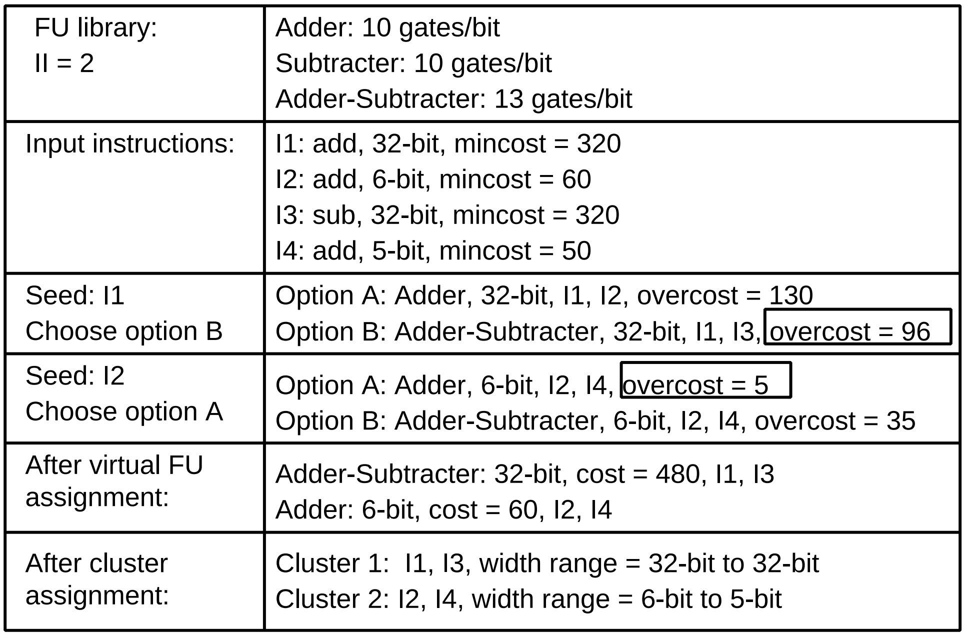
Width clustering allows us to systematically reduce hardware cost by taking advantage of width information without increasing the complexity of FU allocation and scheduling. In fact, width clustering simplifies both the FU allocation and scheduling process. Since FU allocation is performed separately for each cluster, the allocator solves a simpler allocation problem for each cluster. This accelerates the allocation process. Since, the software pipeliner is constrained to bind each operation to scheduling alternatives within its cluster, the number of allowed alternatives is reduced. Again, scheduling is actually simplified by width clustering.

4.4 Example

To illustrate the application of width clustering, the example in Figure 7 is presented. For this example, we assume MAX_OVERCOST is infinity, thus the rapid heuristic for VFU assignment is exclusively utilized. The example consists of four operations, three adds and a subtract, and an II of two. The example FU library has three elements: adder, subtracter, adder-subtracter. The operations are sorted by their inherent cost, yielding an order of I1-I3-I2-I4. The first seed is the head of the list or I1. It can be implemented using either an adder (option A) or an adder-subtracter (option B). With option A, the highest cost operation that is compatible is I2, yielding a overcost of: $(320 - ((320 + 60)/2)) = 130$. With option B, the highest cost operation that is compatible is I3, yielding a overcost of: $(416 - ((320 + 320)/2)) = 96$. The choice with the smallest overcost is chosen; hence, option B is selected. The next seed chosen is I2, and with a similar calculation, option A is chosen. After VFU assignment is complete, there are two VFUs: a 32-bit adder-subtracter assigned operations I1 and I3; and a 6-bit adder assigned operations I2 and I4.

VFU clustering is then performed. Assuming a cluster ratio of two, each VFU is assigned its own cluster. Hence after width clustering is complete, there are two clusters, (I1,I3) and (I2,I4). The creation of the clustered machine description selects an adder-subtracter for the first cluster and an adder for the second cluster. For this simple example, integer linear programming happens to select the same FUs as those that were selected during VFU assignment. Subsequent software pipelining ensures that I1 and I3 are bound to the resources in the first cluster (adder-subtracter) and I2 and I4 are bound to the resources in the second (adder).

It is interesting to re-examine the example with one small change to the FU library. Assume that the cost of the adder-subtracter is increased from thirteen gates/bit to fifteen gates/bit.



FU library: II = 2	Adder: 10 gates/bit Subtracter: 10 gates/bit Adder-Subtracter: 13 gates/bit
Input instructions:	I1: add, 32-bit, mincost = 320 I2: add, 6-bit, mincost = 60 I3: sub, 32-bit, mincost = 320 I4: add, 5-bit, mincost = 50
Seed: I1 Choose option B	Option A: Adder, 32-bit, I1, I2, overcost = 130 Option B: Adder-Subtracter, 32-bit, I1, I3, overcost = 96
Seed: I2 Choose option A	Option A: Adder, 6-bit, I2, I4, overcost = 5 Option B: Adder-Subtracter, 6-bit, I2, I4, overcost = 35
After virtual FU assignment:	Adder-Subtracter: 32-bit, cost = 480, I1, I3 Adder: 6-bit, cost = 60, I2, I4
After cluster assignment:	Cluster 1: I1, I3, width range = 32-bit to 32-bit Cluster 2: I2, I4, width range = 6-bit to 5-bit

Figure 7: Example application of width clustering using rapid heuristic.

In this case, VFU assignment using the rapid heuristic fails to achieve an efficient solution. The result is that three VFUs (32-bit adder, 32-bit subtracter, and 5-bit adder) are assigned operations. Even with the change in cost, the best solution is still two VFUs (32-bit adder-subtracter and 6-bit adder) as achieved previously. The rapid heuristic made an inefficient choice for the first operation assigning it to an adder (rather than an adder-subtracter), thereby causing the problem. The exponential heuristic achieves the best solution for this example for either cost function.

5 Hardware Generation

The final phase of the design process is to build the actual NPA hardware. A hardware processing engine is synthesized directly from the scheduled loop. Each hardware component (FU, register, MUX) is sized using the results from bitwidth analysis and scheduling.

The datapath schema for each processor in the NPA is shown in Figure 8. The datapath consists of an array of heterogeneous FUs that implement all operations in the loop body.

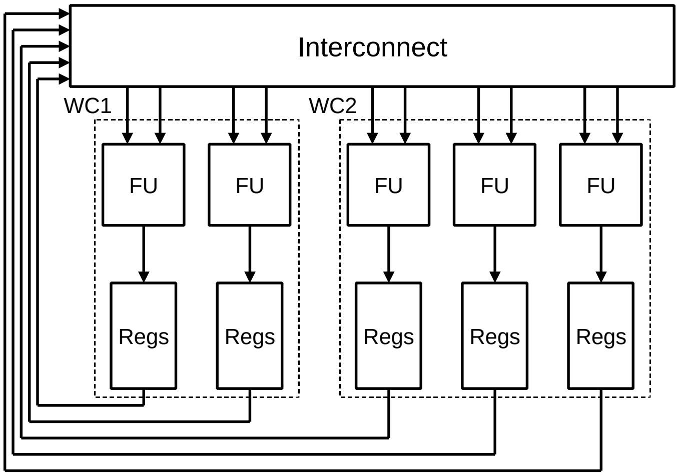


Figure 8: Non-programmable accelerator datapath schema used by PICO.

FUs include adders, multipliers, multiply-adders, ALUs, etc. Ports to memories are treated as FUs as well. The physical memories and memory interfaces are not shown. There is also a special branch FU that controls the software pipeline loop execution [9].

Each FU computes result operands that must be stored in registers until they are no longer needed. Our approach for deploying registers is too complex to fully describe within this paper but a brief overview is presented here. A separate set of registers is dedicated to storing results that are computed within each FU. However, special treatment is needed for rarely occurring cases where, due to the use of predicated conditionals, a common result is computed by multiple FUs within mutually exclusive conditional clauses. Each FU's result registers are implemented as a customized network of individual register elements rather than as a multi-ported addressable register file. The number of required registers depends upon the number of program variables computed by each FU as well as the length of time that each computed value must be maintained to support the software pipeline schedule. After the register network and the flow of operands through registers is fully specified, each register element is further customized to its final width. Before the loop can begin execution, all live-in values are downloaded from the global memory and stored into the appropriate register to initialize the loop.

Because a common set of registers stores all results that are computed within each FU, and because registers often hold values for more than one program variable, width clustering simultaneously reduces the hardware cost for the FUs as well as the cost of the FUs' result registers. That is, the clustering of operations of similar width into common FUs automatically clusters operands of similar width into shared register elements.

Loop invariant operands receive special treatment. Constant values are directly generated in hardware. Loop invariant values that are computed prior to entering the loop and then repeatedly used within the loop require exactly one unshared register.

The datapath is controlled by a ring counter that varies from 0 to II-1 and a loop counter that is initialized to the number of loop iterations and decremented until it reaches 0. The ring counter is used to generated control signals for switches within the interconnect, registers, and multi-function FUs. A final DONE flag is set when the desired number of iterations have been executed and the pipeline is drained.

6 Experimental Evaluation

In this section, PICO's bitwidth-sensitive architecture synthesis is evaluated. The comparison is made against a baseline PICO that is bitwidth unaware.

6.1 Setup and application characteristics

To perform the experiments, we used PICO to design NPAs for a set of twenty loop nests. Table 2 presents the loop nests and a brief description of each. The depth of each loop nest in the original source is specified in the column labeled Depth. These loop nests were chosen from a variety of domains including printing, digital photography, communications, and networking. Narrow bitwidths are common in these domains and used throughout these applications. Width pragmas were inserted where appropriate to more precisely specify the widths of values kept in memory (e.g., arrays).

Table 2: Application description and target throughput.

Application	Depth	II	Description
adpcm	2	9	adaptive speech compression
cell	2	2	packet recognition and delineation
chain	2	2	synthetic benchmark
channel	2	17	multiplexing cells on a channel
conv2d	4	3	2D convolution
dct	2	2	forward discrete cosine transform
edge	2	2	edge-based image smoothing
encode	2	2	run-length encoding
fir	2	2	16-tap finite impulse response filter
fsed	2	3	Floyd-Steinberg halftoning
heat	2	2	1D relaxation
huffman	2	2	huffman encoding
linescreen	2	2	image half-toning
lyapunov	3	3	stablility analysis
matmul	3	2	matrix multiplication
rls	3	2	complex recursive least-squares filter
sharp	2	2	image sharpening
sobel	2	2	image edge detection
taub	6	2	digital camera demosaicing
viterbi	2	6	viterbi decoder using block decoding

One application, *chain*, is a synthetic application that was created during our study of width-aware synthesis. *Chain* is a loop nest that contains two dependence chains of multiply operations that are identical except in the width of data they process. The first chain operates on narrow data and the second on wide data. In such an application, the opportunity for large cost savings using bitwidth analysis is present because half of the data is narrow. However, without width-aware heuristics, most FUs end up being wide due to the unfortunate binding of wide and narrow operations to the same FU.

For these experiments, the performance is held constant for each loop nest as specified by the II and the number of processors. For each loop nest, the number of processors is set to one and the chosen II is shown in Table 2. Scaling the number of processors should have little effect on the results, because at higher throughputs identical processors are replicated. By default, an II of two was chosen. However, there were several cases that contained a recurrence constraint that requires an II that is larger than two. For these loop nests, the lowest II that met the recurrence constraint was chosen. The figure of merit in these experiments is the cost of the design that achieves the specified performance. PICO measures cost using gate count estimates for each hardware component. Each component has an associated parameterized cost formula that has been calibrated against a production-quality design library. To derive the total cost, the hardware components are instantiated and the cost of the components are summed across the design. These cost formulas have been shown to accurately estimate system cost as measured in gate equivalents. Cost estimates do not include the cost of wires (including their length).

The width clustering algorithm presented in Section 4 provides a MAX_OVERCOST parameter to determine the heuristic that is applied for virtual FU assignment. Except for the last experiment, these experiments are performed with MAX_OVERCOST set to infinity. This causes the rapid, non-recursive algorithm to be used for these experiments.

To provide some insight into the width characteristics of the applications, a histogram of the static operation widths is presented in Table 3. Each cell in the table contains the fraction of static operations for a particular application whose width is within the specified range. For example, 27% of the operations in *adpcm* have widths of 1-4 bits. As previously discussed, this paper makes the simplifying assumption that an operation can be described by a single width that corresponds to the maximum width across all of its input and output ports. In general, a diverse set of widths are present in each application. Most applications also contain a large fraction of operations whose width is less than 8 bits. A notable exception to these trends is *matmul*. This application is a matrix multiplication of two 32-bit matrices of large size, thus all variables are truly 32 bits. One can properly anticipate that width-sensitive synthesis will have little effect on *matmul* due to this characteristic. In the remainder of the applications, many of the 32-bit operations correspond to address calculation and manipulation. We currently assume all loads and stores to global memory require 32-bit addresses. Thus, in many cases, further improvement can be obtained for the bitwidth of address arithmetic.

Table 3: Distribution of static operation widths.

Application	1-4	5-8	9-12	13-16	17-20	21-24	25-28	29-32
adpcm	0.27	0.15	0.03	0.25	0.14	0.00	0.00	0.15
cell	0.89	0.04	0.00	0.05	0.00	0.00	0.00	0.02
chain	0.43	0.00	0.00	0.00	0.00	0.00	0.00	0.57
channel	0.26	0.15	0.00	0.03	0.02	0.00	0.00	0.55
conv2d	0.52	0.12	0.03	0.00	0.03	0.00	0.00	0.30
dct	0.06	0.01	0.21	0.33	0.00	0.00	0.01	0.38
edge	0.37	0.23	0.20	0.02	0.09	0.03	0.00	0.05
encode	0.09	0.22	0.02	0.24	0.05	0.00	0.00	0.38
fir	0.00	0.25	0.00	0.50	0.00	0.00	0.00	0.25
fsed	0.39	0.22	0.31	0.00	0.00	0.00	0.00	0.08
heat	0.45	0.00	0.21	0.00	0.00	0.00	0.00	0.34
huffman	0.23	0.48	0.09	0.05	0.01	0.01	0.00	0.13
linescreen	0.06	0.63	0.00	0.17	0.00	0.00	0.00	0.14
lyapunov	0.29	0.14	0.06	0.00	0.00	0.00	0.00	0.51
matmul	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00
rls	0.08	0.09	0.00	0.02	0.09	0.00	0.00	0.72
sharp	0.48	0.18	0.09	0.13	0.00	0.00	0.03	0.09
sobel	0.35	0.19	0.29	0.08	0.00	0.02	0.02	0.06
taub	0.38	0.07	0.02	0.27	0.00	0.00	0.14	0.11
viterbi	0.28	0.17	0.30	0.16	0.00	0.00	0.00	0.09

6.2 Effectiveness of bitwidth analysis

Figure 9 presents the effects of bitwidth analysis on the NPA cost for each application along with the arithmetic mean (amean) across all of the applications. The figure compares two variants of the PICO-NPA system: no width cognizance where the standard C widths are used for all variables and operations (left bar), bitwidth analysis enabled but width clustering disabled (right bar). The bars show the normalized cost for each NPA design broken down into three pieces: FU, register, and the remainder or rest. The remainder portion is dominated by switches within the interconnect of the design (see Figure 8). As with FUs and registers, the cost of the interconnect is highly dependent on width. Total cost for each bar is normalized to the no width cognizance case, thus the height of the rightmost bar shows the overall cost reduction achieved via bitwidth analysis.

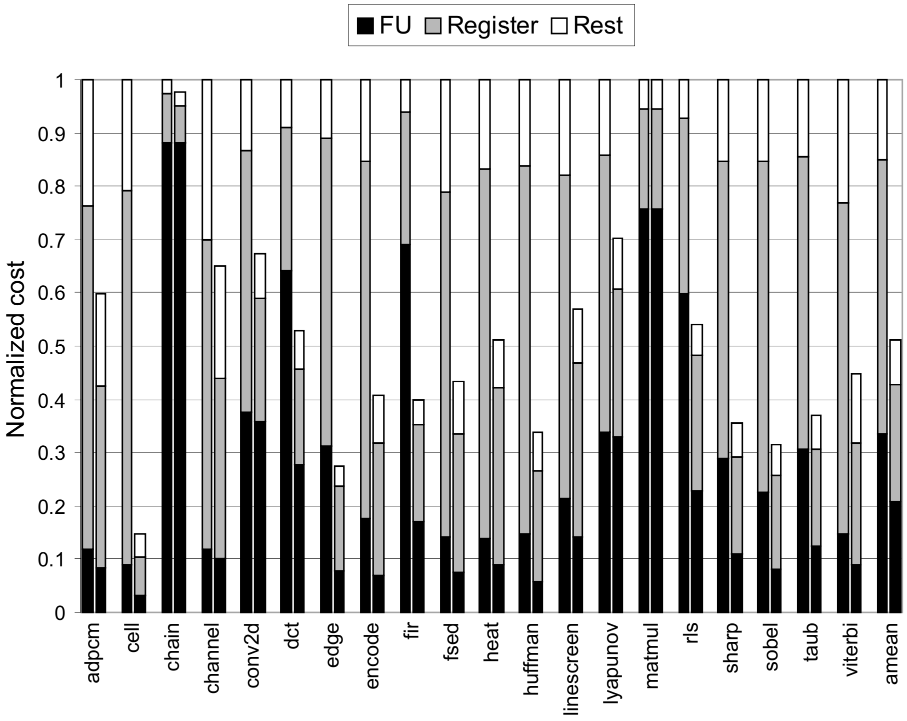


Figure 9: Effects of bitwidth analysis on NPA cost. The study compares two configurations to determine component widths: standard C widths (left bar) and bitwidth analysis (right bar). Cost is broken down into three pieces: FU, register, and rest.

From the figure, bitwidth analysis alone provides a large reduction in total cost across most of the loops. The mean total cost is reduced by approximately 50%. This is achieved by reducing the mean costs of the FUs, registers, and rest by 38%, 57%, and 45%, respectively. Interestingly, the register cost is reduced by the largest percentage and the FU cost by the smallest. A common cause for this behavior is that many of the loops contain uniformly wide multiply operations. Multipliers are quadratic whereas registers are linear in cost as a function of width. As a result, the FU cost has an expensive fixed term due to wide multipliers. Therefore, a smaller reduction is observed for a very expensive term in the FU cost.

The largest reduction occurs for *cell*, where the total cost is reduced by 85%. This application is dominated by operations that are 1-4 bits (89% from Table 3). Hence, there are a large number of opportunities to synthesize narrow hardware to reduce cost. There are few problems that arise due to any sharing of hardware between wide and narrow operations. The other extreme behavior occurs for *matmul*, where no cost reduction is observed. As shown in Table 3, all of its operations are 32 bit, hence there is no opportunity for bitwidth-sensitive synthesis to yield any cost reductions.

6.3 Effectiveness of width clustering

Figure 10 presents the effects of width clustering on the NPA cost for each application along with the arithmetic mean across all of the applications. The format of the figure is identical to that of the previous experiment (Figure 9). However, Figure 10 compares two different variants of the PICO-NPA system: bitwidth analysis alone (left bar) and bitwidth analysis and width clustering (right bar). Total cost for each bar is normalized to the bitwidth analysis alone case.

The figure shows that width clustering further improves to the cost of the NPAs, but the improvement is more modest than in the previous experiment. A mean reduction of 9% in total cost is observed, achieved by mean reductions of 11% FU, 6% register, and 9% rest. The most noticeable cost savings occurs for *chain*, which enjoys a 45% reduction in total cost. This application suffers from poor sharing of hardware using bitwidth analysis alone. Almost half of the operations in *chain* are 1-4 bits (see Table 3), yet less than a 5% reduction in cost is observed after bitwidth analysis (see Figure 9). Narrow and wide operations are accidentally scheduled onto common FUs resulting in most of the hardware being wide. Width clustering effectively groups narrow operations together, thereby reducing the FU cost by a substantial amount. A similar behavior occurs for *lyapunov* and yields a total cost reduction of 19%.

Other applications, which achieve more than 10% reduction in total cost via width clustering, illustrate a different behavior. Two such examples are *channel* and *encode*. In both of these cases, the FU cost drops by less than 5%. However, the register cost drops by 21% and 15%. Both of these designs are dominated by register cost, because there are a large number

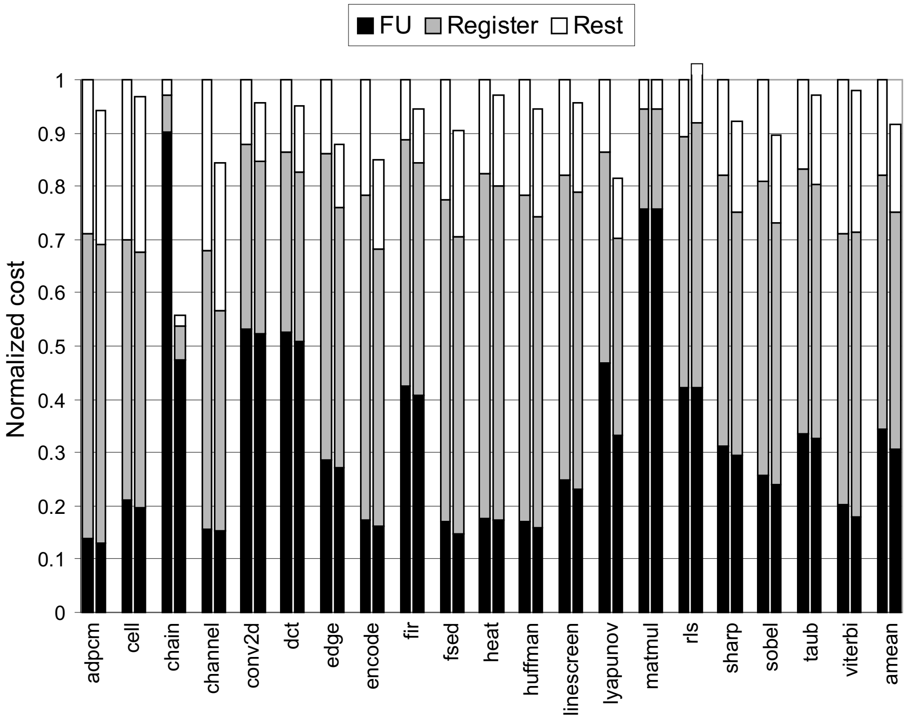


Figure 10: Effects of width clustering on NPA cost. The study compares two configurations to determine component widths: bitwidth analysis alone (left bar) and bitwidth analysis and width clustering (right bar). Cost is broken down into three pieces: FU, register, and rest.

of variables with long lifetimes. Without width clustering, wide and narrow operations are placed on common FUs. This binding results in wide FUs, but more importantly results in wide registers because our datapath schema shares registers among the values produced by a single FU (see Section 5). Width clustering is able to effectively group operations of similar width to enable the width of the output registers to be substantially reduced. FU cost is also reduced, but the amount is insignificant compared to the savings in the register cost.

The *rls* application is an outlier. In this case, width clustering increases the cost of the NPA by 3%. This behavior results because width clustering causes the schedule length for a single iteration of the loop to increase. We believe that this is because the scheduler has fewer binding choices due to the clustering, and must lengthen the schedule to achieve the desired II. The net effect is that a larger number of registers is required and the cost grows slightly.

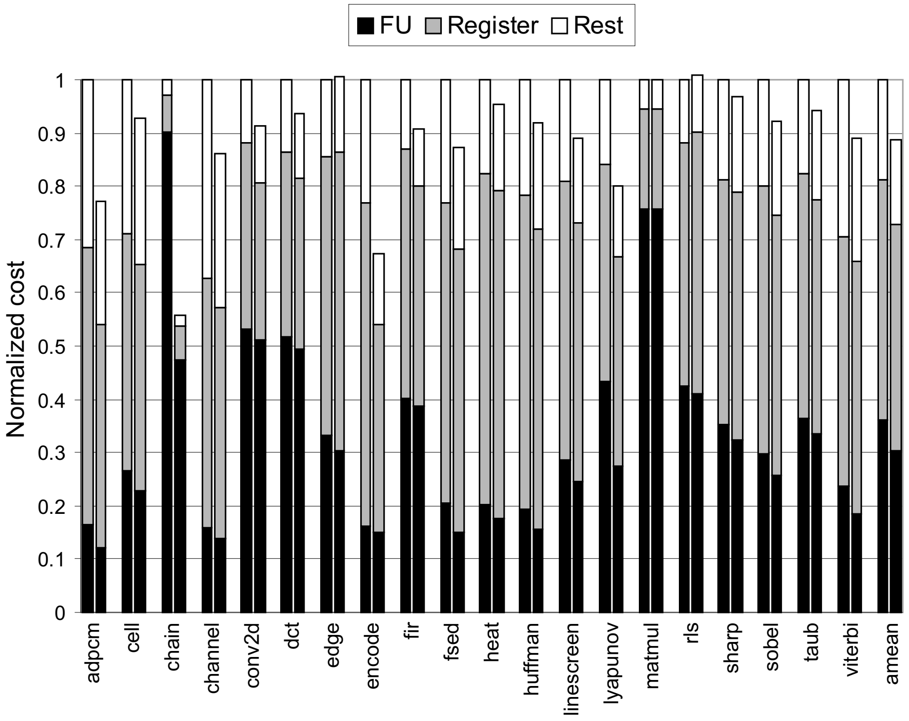


Figure 11: Effects of width clustering on NPA cost using a richer FU library that contains multi-function FUs. The study compares two configurations to determine component widths: bitwidth analysis alone (left bar) and bitwidth analysis and width clustering (right bar). Cost is broken down into three pieces: FU, register, and rest.

6.4 Width clustering with multi-function FUs

One of the key factors affecting the results is the set of FUs available in the library. The baseline library in PICO supports only a small number of multi-function FUs, such as adder-subtracter, multiply-adder, and load-store. Multi-function FUs create opportunities for intelligent sharing that can be exploited by width clustering. To investigate these effects, the last experiment is repeated with a number of multi-function FUs added to the PICO library. The results of the experiment are presented in Figure 11. The figure has the same format as Figure 10. However, note that the cost is normalized to a different value in this experiment due to the different FU library.

The results in Figure 11 are noticeably different from those in Figure 10. The mean reduction in total cost increases from 9% to 12%. This behavior is directly attributable to the availability of multi-function FUs. Multi-function FUs support larger combinations of operation types. Hence, there are more interesting sharing opportunities to exploit during width clustering. The flexibility of multi-function FUs enables the mapping of more operations of similar width to common FUs when it has little effect on the cost of the FU. In addition to reducing overall FU cost, this behavior generally reduces the cost of registers and interconnect as better width utilization is achieved for the entire datapath. One obvious example of this behavior is *adpcm*. In this case, a 23% reduction in total cost is achieved via width clustering compared to 6% reduction in the previous experiment without multi-function FUs.

There is one outlying application, *edge*, where the relative cost of the NPA is increased with multi-function FUs and width clustering. For this application, width clustering provides a 12% reduction in cost using the base FU library (Figure 10). However, using multi-function FUs, width clustering increases the cost by 1%. The single-iteration schedule length is increased by a significant amount by the choice of clusters. As a result, the register and total cost also grow.

6.5 Comparison of width clustering heuristics

The width clustering algorithm presented in Section 4 contains two separate heuristics for the VFU assignment phase. The rapid heuristic employs the overcost metric to make assignment decisions. There is also a more expensive heuristic that employs a fully-recursive technique to derive the VFU assignments. Using both FU libraries from the previous experiments, we compared the gate counts achieved with exclusive use of each heuristic. The results showed only minor differences in the achieved FU and total gate counts. They differed by no more than 3%. Generally, the expensive heuristic achieved better results, but there were several cases where the rapid actually performed better. In many cases, the results were virtually identical.

From these results, one might conclude that the expensive heuristic is not needed. We believe, however, that such a conclusion cannot be sustained at this point for a number of reasons. First, we only evaluated a small number of applications for these experiments. Second, PICO's FU library is less complex than what we expect to encounter in fully practical uses. Third, it is not difficult to break the rapid heuristic, as shown in Section 4.4. A production architecture synthesis system is likely to face more complex applications and a richer FU library. In such an environment, the tradeoffs are more difficult, and we believe the expensive heuristic may have be better than this set of experiments shows.

7 Related Work

Bitwidth has been exploited in a number of previous efforts. The C language has been augmented to provide additional bitwidth information in the work on Valen-C at Kyushu University [10], and by using pragmas in work at Delft [11]. Our C extensions closely mirror these pragmas. A number of prior efforts propagate bitwidth information in the style of dataflow analysis. Information is propagated only locally in [11]. Others propagate information over larger scope [12] [13] [14]. This work is similar to ours in that we all use bidirectional constraint propagation. The work at MIT [14] emphasizes the careful treatment of value ranges, while the work at CMU [13] analyzes sparse patterns of bits by recording detailed information about each bit position separately. Each of these bitwidth analysis approaches can potentially discover opportunities that are missed by our analysis approach.

In work at Seoul National University [15] [16], the effects of quantization error for fixed point operations where low order bits are discarded is studied using both analysis and simulation. This work treats a limited class of add and multiply based signal processing algorithms. While our approach never sacrifices any precision, it is clear that for many digital signal processing applications low-order bits are often not needed and can be discarded in order to reduce hardware cost without introducing undue error into the application.

Automatic datapath synthesis and has a long history and vast literature. For example, Cathedral III [17], represents a complete synthesis system developed at IMEC and illustrates one approach to high-level synthesis. It uses an applicative language for program specification and designs customized datapaths for DSP applications from this specification.

Our work focuses specifically on datapath synthesis in the context of $II \geq 1$ software pipelines that share resources among multiple operations. This requires both the allocation of hardware resources as well as the scheduling of operations to those resources. The focus on software pipelines allows us to allocate hardware using resource models that have been carefully adapted to software pipelining. We do not know of other datapath synthesis systems that generate low-cost designs by scheduling loops at a desired throughput on FUs that are shared among operations of similar width in such a way as to reduce hardware cost.

Paulin and Knight use a technique called force-directed scheduling to synthesize datapaths in the HAL system for ASIC design [18]. They integrate FU resource allocation and scheduling into a common synthesis algorithm to minimize overall cost. The Sehwa design system automatically designs processing pipelines from behavioral specifications [19]. This work uses allocation and scheduling heuristics to construct cost or performance constrained designs. Bakshi and Gajski consider the tradeoffs in allocating either low latency and expensive or high latency and inexpensive FUs within an integrated scheduling and resource allocation algorithm [20]. Similarly, Chang and Pedram also consider the allocation of FUs of varying latency but their focus is on energy minimization [21]. Clique based partitioning algorithms were developed in the FACET project to jointly minimize FU and inter-FU communication costs [22]. In [16], greedy list scheduling techniques are presented that use bitwidth information during scheduling to select hardware units having compatible width. An additional problem of minimizing the cost of transmitting and extending operands of variable bitwidth has been addressed in [23].

Marwedel studies techniques that allow the use of common hardware to treat expressions with related, but not identical, semantics [24]. The technique consists of an initial phase that maps expressions to virtual components followed by a subsequent phase that maps virtual components to physical components. Ang and Dutt develop techniques to optimize multi-output operations. They also consider a simple linear-cost treatment for bitwidth [25].

Another approach customizes a conventional processor with respect to bitwidth. In work by Shackleford *et al.* detailed bitwidth information on operations is used to explore the cost effectiveness of a family of processors with varying datapath width [26]. When operands have width that exceeds the hardware width, they are treated in a serial fashion using multiple precision operations. As the hardware width is varied, bitwidth information on operations allows the system to determine the precise number of computational steps required for each operation.

Scheduling within clusters has been used for VLIW architectures that are implemented as separate physical clusters [8] [27] [28] [29] [30]. These clustering heuristics are aimed at compilation for predefined VLIW architectures that have partitioned FUs and register files. In these machines, inter-cluster communication is costly and may require the insertion of inter-cluster copy operations. The goal is to intelligently assign operations and operands to clusters so that performance is maximized.

Eijk *et al.* [31] present an approach for scheduling code for irregular CPUs. This work deals with complex machine constraints by pruning the search space prior to scheduling. While the problem they solve is quite different from ours, they also use an approach that limits the binding choices before scheduling starts.

8 Conclusion

In this paper, we investigate the exploitation of integer bitwidth in an architecture synthesis system for custom nonprogrammable hardware accelerators. The goal is to reduce the cost of our designs by exploiting bitwidth information to build cheaper hardware. We employ two complementary approaches. Bitwidth analysis computes the number of bits necessary for each program variable and operation. This information provides the foundation for architecture synthesis. Width clustering is then used to guide FU allocation and instruction scheduling so that they intelligently map operations of disparate bitwidths onto the hardware. Sharing decisions are made jointly based on bitwidth and implementation cost.

Experiments show that bitwidth-sensitive architecture synthesis reduces design cost by a substantial amount. Bitwidth analysis alone provides a mean reduction in total gate count of 49%. The application of width clustering provides an additional reduction of 9% in mean total gates. Overall, the mean design cost is reduced by 53% over a baseline system that is bitwidth unaware. The experiments also show that the importance of width clustering increases as the number of architectural choices increases. The scheduler is more prone to making bad decisions concerning width, resulting in poor designs. Width clustering effectively constrains the scheduling choices to produce a quality design.

This work is based on a number of assumptions that could be generalized in future research. The current synthesis system assumes that all FUs are fully pipelined and can process a new set of input operands on every clock cycle. This limitation can be eliminated using relatively simple techniques to model, allocate, and synthesize operations that occupy FUs for multiple machine cycles. Results presented here also assume that each operation's latency is known prior to FU binding. That is, for each operation, the binding choice is limited to a set of FUs having common latency. This too can be generalized; however, problems occur when operations on recurrence cycles are chosen with excessive latency and the scheduler fails to identify a legal program schedule. Finally, the work also assumes that, for each operation, all input and output operands have the same bitwidth. For operations like shifts and multiplies, this is a severe limitation. Removing this limitation is somewhat more difficult because of the increased complexity in modeling cost and the large number of binding choices that are now presented. For example, when treating a mixed set of commutative and non-commutative operations, hardware optimization should consider applying the commutative property, where legal, to co-align narrow operands on the same side of each potential FU. Improvements in all of these areas would make this work more generally applicable to future design systems.

Acknowledgments

The authors thank Santosh Abraham for his help in designing and developing bitwidth analysis; Shail Aditya, Darren Cronquist, Vinod Kathail, Bob Rau, and Mukund Sivaraman for their many discussions and useful feedback.

References

[1] R. Schreiber *et al.*, "High-level synthesis of nonprogrammable hardware accelerators," in *Proceedings of the International Conference on Application-Specific Systems, Architectures, and Processors* (E. E. Swartzlander, G. A. Jullian, and M. J. Schulte, eds.), pp. 113–124, July 2000.

[2] S. Aditya, B. R. Rau, and V. Kathail, "Automatic architectural synthesis of VLIW and EPIC processors," in *International Symposium on System Synthesis, ISSS'99*, pp. 107–113, Nov. 1999.

[3] *The Trimaran Compiler Infrastructure for Instruction-Level Parallelism.* www.trimaran.org.

[4] R. Schreiber *et al.*, "PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators," *Journal of VLSI Signal Processing*, vol. to appear, 2001.

[5] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.

[6] C. H. Gebotys and M. I. Elmasry, "Global optimization approach for architecture synthesis," *IEEE Transactions on Computer Aided Design*, 1993.

[7] B. Landwehr, P. Marwedel, and R. Domer, "Oscar: Optimum simultaneous scheduling, allocation and resource binding based on integer programming," in *IFIP Workshop on Logic and Architecture Synthesis*, (Grenoble), 1994.

[8] P. Lowney *et al.*, "The Multiflow Trace scheduling compiler," *The Journal of Supercomputing*, vol. 7, pp. 51–142, Jan. 1993.

[9] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL PlayDoh architecture specification: Version 1.0," Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, Palo Alto, CA 94304, Feb. 1994.

[10] H. Yasuura, H. Tomiyama, A. Inoue, and F. N. Eko, "Embedded system design using soft-core processor and Valen-C," *IIS Journal of Information Science and Engineering*, vol. 14, pp. 587–603, Sept. 1998.

[11] A. Cilio and H. Corporaal, "Efficient code generation for ASIPs with different word sizes," in *Third Annual Conference of the Advance School for Computing and Imaging*, (The Netherlands), June 1997.

[12] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable function units," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pp. 172–180, Nov. 1994.

[13] M. Budiu, S. Goldstein, K. Walker, and M. Sakr, "Bitvalue inference: Detecting and exploiting narrow bitwidth computations," in *Euro-Par 2000 Parallel Processing* (A. Bode, T. Ludwig, W. Karl, and R. Wismüller, eds.), vol. 1900 of *Lecture Notes In Computer Science*, pp. 969–979, Springer-Verlag, 2000.

[14] M. Stephenson, J. Babb, and S. Amarasinghe, "Bitwidth analysis with application to silicon compilation," in *Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, pp. 108–120, June 2000.

[15] S. Lee and W. Sung, "Finite wordlength effects analysis and wordlength optimization of Dolby digital audio decoder," in *Proceedings of ISCAS'98*, June 1998.

[16] K. I. Kum and W. Sung, "Word-length optimization for high level synthesis of digital signal processing systems," in *Proceedings of 1998 IEEE Workshop on Signal Processing Systems*, pp. 142–151, Oct. 1998.

[17] S. Note, W. Geurts, F. Catthoor, and H. D. Man, "Cathedral-III: Architecture-driven high-level synthesis for high throughput DSP applications," in *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pp. 597–602, June 1991.

[18] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavorial synthesis of ASIC's," *IEEE Transactions on Computer-Aided Design*, vol. 8, pp. 661–679, June 1989.

[19] N. Park and A. C. Parker, "Sehwa: A software package for synthesis of pipelines from behavioral specifications," *IEEE Transactions on Computer-Aided Design*, vol. 7, pp. 356–370, Mar. 1988.

[20] S. Bakshi and D. D. Gajski, "Components selection for high performance pipelines," *IEEE Transactions on VLSI Systems*, vol. 4, pp. 182–194, June 1996.

[21] J. M. Chang and M. Pedram, "Energy minimization using multiple supply voltages," *IEEE Transactions on VLSI Systems*, vol. 5, pp. 1–8, Dec. 1997.

[22] C. Tseng and D. P. Siewiorek, "FACET: A procedure for automated synthesis of digital systems," in *Proceedings of the 20th Design Automation Conference*, pp. 566–572, June 1983.

[23] K. Schoofs, G. Goossens, and H. D. Man, "Bit-alignment in hardware allocation for multiplexed DSP architectures," in *Proceedings of the European Conference on Design Automation*, pp. 289–293, Feb. 1993.

[24] P. Marwedel, "Matching system and component behaviour in MIMOLA synthesis tools," in *Proceedings of the European Design Automation Conference*, Mar. 1990.

[25] R. Ang and N. Dutt, "An algorithm for the allocation of functional units from realistic RT component libraries," in *Proceedings of the Seventh International Symposium on High-level Synthesis*, pp. 164–169, May 1994.

[26] B. Shackleford *et al.*, "Embedded system cost optimization via data path width adjustment," *IEICE Transactions on Information and Systems*, vol. E80-D, pp. 974–981, Oct. 1997.

[27] A. Capitanio, N. Dutt, and A. Nicolau, "Partitioned register files for VLIWs: A preliminary analysis," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 292–300, Dec. 1992.

[28] W. Lee *et al.*, "Space-time scheduling of instruction-level parallelism on a RAW machine," in *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 46–57, Oct. 1998.

[29] E. Ozer, S. Banerjia, and T. M. Conte, "Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures," in *Proceedings of the 31th Annual International Symposium on Microarchitecture*, pp. 308–314, Nov. 1998.

[30] G. Desoli, "Instruction assignment for clustered VLIW DSP compilers: A new approach," Tech. Rep. HPL-98-13, Hewlett-Packard Laboratories, Palo Alto, CA 94304, Feb. 1999.

[31] K. Eijk *et al.*, "Constraint analysis for code generation: Basic techniques and applications in FACTS," *ACM Transactions on Design Automation of Electronic Systems*, vol. 5, pp. 774–793, Oct. 2000.