Bitwidth Sensitive Code Generation in a Custom Embedded Accelerator Design System

Scott Mahlke Timothy Sherwood Rajiv Ravindran Mike Schlansker Rob Schreiber
Hewlett-Packard Laboratories, Palo Alto, CA

1 Introduction

An ever larger variety of embedded ASICs is being designed and deployed to satisfy an explosively growing demand for new electronic devices. Many of these devices are called upon for computationally demanding processing of multimedia data. In many such ASICs, specialized nonprogrammable hardware accelerators (NPAs) are used for 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, and high performance are all 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 highly parallel, special-purpose processor array. The user, or a design exploration tool, specifies the number of processing elements and their performance. The system produces a VHDL design for the array, its control logic, its interface to memory, and its interface to a host processor.

The quality of automatically generated designs is an important issue. In the multimedia application domain, integer data of various *bitwidths* is common (e.g., 8-bit and 16-bit data). The bitwidth is defined as the number of bits required to represent the range of possible values in two's complemement. Thus, one large opportunity for improving design quality is to take advantage of operations with varying bitwidths. The width of hardware components, including registers, busses, and function units, can be optimized by taking advantage of the maximum width of data processed by the component.

In this paper, we investigate the exploitation of integer bitwidth in the code generation system of an NPA compiler. The code generation consists of two major parts. First, a bitwidth analysis phase is used to infer the bitwidth requirements for all program data. We allow users to define the bitwidth of selected variables through declarations in the source code. With this information as the starting point, the bitwidth analysis examines the data dependence structure of the application together with opcode semantics to derive the minimum width required for all program variables.

The second part of the NPA code generation system handles the mapping of operations to resources. More specifically, this part includes the allocation of hardware resources and scheduling where operations are assigned to those resources. With operations of varying bitwidths, the mapping must be done in a width-cognizant manner to achieve a cost-effective design. Intuitively, operations of similar width should be assigned to common hardware resources, so narrow resources are used for operations with narrow data and wide resources are used for operations with wide data. The goal is to minimize the overall cost of the hardware resources. We employ a technique referred to as *width clustering*, in which operations with similar bitwidths are grouped into clusters before scheduling. Hardware resources are then allocated separately for each cluster. During scheduling, operations are restricted in their binding to resources from their own cluster. In this manner, the operation binding choices are limited in order to encourage intelligent grouping of operations to resources based on bitwidth information.

The remainder of this paper provides an overview of the PICO NPA design system. Then, a description of the two major components of the code generation system that handle bitwidth specific issues are described. Last, some

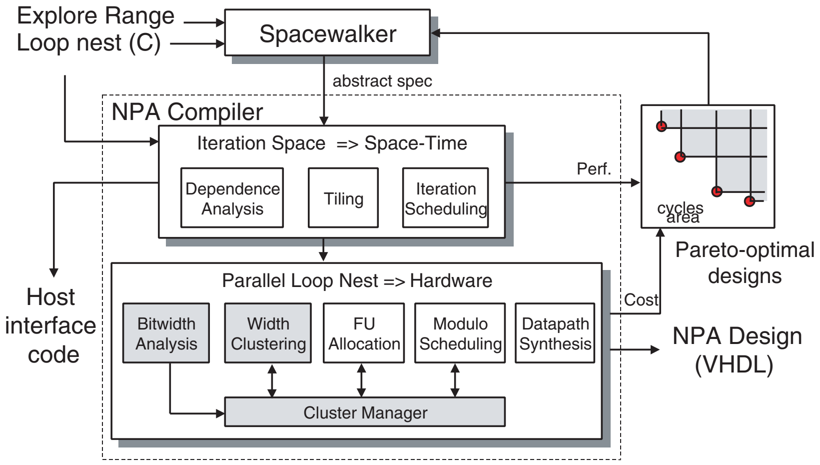

Figure 1: PICO hardware accelerator design system.

ongoing efforts in this work are provided.

2 Overview of the PICO Design System

The overall structure of the PICO design system is shown in Figure 1. The user provides a C loop nest as well as a range of architectures to be explored. The *spacewalker* is responsible for finding the best designs, i.e., the points such that no explored design has both lower cost and better performance. The spacewalker specifies a processor count, a performance per processor, and a global memory bandwidth to the NPA compiler. The NPA compiler is responsible for creating (and expressing in VHDL) an efficient, detailed NPA design for the given loop nest, consistent with the abstract architecture specification provided by the spacewalker. It also generates an accurate performance measurement and an estimated gate count for the NPA.

The NPA compiler transforms the loop nest into a customized RTL architecture via a process consisting of:

1. An analysis phase in which array accesses and value-based flow dependences are found.
2. A tiling/mapping/scheduling phase in which the tile shape and the mapping of iterations to processors and to clock cycles are determined.
3. A loop transformation phase in which the loop nest is first tiled, the outer loops over tiles are sequential, and the inner loops over iterations within a tile are rewritten into parallel form. This form mirrors the hardware's structure in that the parallel loops are over the processor indices. The loop body is also made to reflect the hardware implementation, through explicit register promotion, load-store minimization, explicit interprocessor communication, explicit data location in global or local memory, and a temporal-recurrence scheme for computing iteration space coordinates, memory addresses, and predicates.
4. A processor synthesis phase in which the processor's computation assets are allocated, the operations and the data of the loop body are bound to these assets and scheduled relative to the start time of the loop (modulo scheduling), and the processor storage assets and intra- and inter-processor interconnects are created.
5. A system synthesis phase in which multiple copies of the processor are allocated and interconnected, and the controller and the data interfaces are designed.

The NPA compiler backend is extended to support width-cognizant code generation and datapath synthesis via the shaded modules in Figure 1. The cluster manager serves as the conduit for all bitwidth information across the system. The starting point consists of an assembly-level dependence graph representing the parallel loop nest. Bitwidth analysis calculates the minimum width for each operand in the dependence graph.

The next step groups operations of similar width into clusters. This cluster information is then used to drive the downstream components of the compiler: FU allocation and modulo scheduling. FU allocation allocates resources for each cluster. The modulo scheduler subsequently binds operations to hardware resources within predetermined clusters and to time slots. The binding performed during scheduling has not been totally predetermined. Rather, the clustering and FU allocation provide a partial binding to guide the scheduler's choice, but the scheduler may choose any compatible resource in the operation's cluster.

3 Bitwidth Analysis

PICO uses two separate, complementary approaches to finding required bitwidths: a priori bounds on values for specific variables, and an iterative constraint propagation analysis that we describe below. In transforming the original program to parallel form, PICO introduces a number of variables into the code. Bounds on their values are generally known at compile time, and their required widths are therefore known. In addition, we allow the user complete control over bitwidths of variables; a pragma containing an arbitrary bitwidth of a variable (e.g., 5 bits) may be optionally supplied after each variable declaration. These bitwidths are passed down automatically through the PICO frontend to help the later iterative constraint propagation phase achieve better results.

The approach used for bitwidth inference is to iteratively propagate 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 variable need not retain any more bits than the number needed by its uses. This is referred to as the *use constraint*. For example, a 32-bit quantity contains an excess of data if it is only used in 10-bit add operations. We may repeatedly apply the def and use constraints to get ever tighter restrictions on variable widths until we converge to a stable solution.

The bitwidth analysis algorithm is composed of two passes corresponding to taking into account each of the width constraints. 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. The first pass, referred to as the forward propagation phase, utilizes the def constraint to limit the output width of each operation. The required width of the output of an operation is determined as a function of the operation being performed (add, multiply, etc.), the current width of the inputs, the value of any literal input, and the semantics of C. For example, when two 6-bit quantities are added, it is known the result is not larger than 7 bits. The values of literal inputs provide key constraints to the output width. For example, the bitwise AND of a 32-bit variable and the literal 0xFF produces an 8-bit result, regardless of the value of the 32-bit variable. The forward propagation phase is applied iteratively across all operations in the program until a fixed point solution is reached.

The second pass of bitwidth analysis, referred to the backward propagation phase, enforces the use constraints to limit the input widths of each operation based on its output widths, and to limit the width of each variable based on the widths of all of its uses. As in the forward propagation phase, input widths are constrained based on the operation being performed, the width of the outputs, the actual value of any literal inputs, and the semantics of C. For example, an add operation with a 15-bit result needs inputs of no more than 15 bits. More interesting, left shift operation with an 8-bit output and shift amount of the literal 3 constrains the other input to be no more than 5 bits. As with the forward propagation phase, the backward phase is applied iteratively across all operations until a fixed point solution is reached.

To illustrate the application of bitwidth analysis, consider the example in Figure 2. The original code consists of 4 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 2-bit quantity, produces at most a 4-bit result, hence the width of x is no more than 4 bits. For I2, the 4-bit value for x is propagated downward from I1. Then, it is known that the multiplication of n bits by m bits yields at most n+m bits. Thus, the width of y is concluded to be no more than 15 bits. Similar propagation is applied to the other instructions. Since I3 is within a loop, the forward propagation will

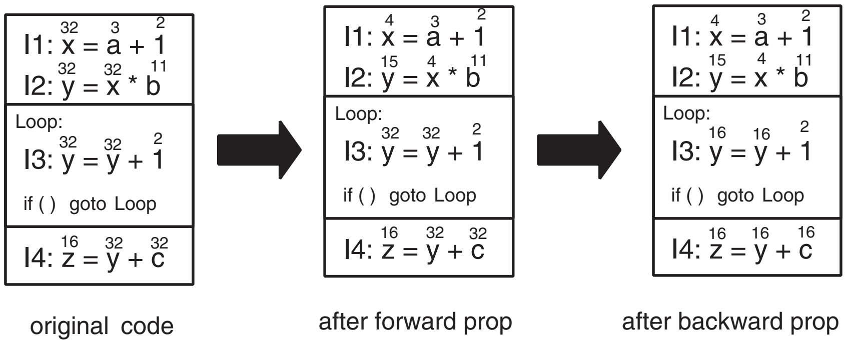

Figure 2: Example application of bitwidth analysis.

iterate until a fixed point solution is reached which concludes y could be 32 bits. This result makes sense as the loop iterates an unknown number of times.

Backward propagation is applied next. The constraint of the final output, z, being no more than 16 bits is propagated. This effects the width of y and c in I4 and I3 because the 16-bit output requires only 16-bit inputs be available. I1 and I2 are not affected by the backward propagation because they already contain stronger width constraints.

4 Width Clustering

The goal of width clustering is to partition a set of operations into groups of similar bitwidth. A straight-forward approach would perform a standard clustering algorithm on the bitwidths of all the operations. However, this approach is undesirable because it does not take into account the eventual assignment of operations to function units (FUs), nor the cost of the FUs. Thus, expensive FUs (e.g., divide) must often be unnecessarily replicated. To avoid this problem, we first perform a simple binding of operations to FUs. This assignment is not used by the rest of the system, hence this process is called virtual FU assignment. But, it provides a sample mapping from which more intelligent clustering decisions can be made. The virtual FUs are then clustered together based on the FU width, which is determined by the widest operation assigned to that unit. The result is operations are partitioned into clusters based on a combination of width and FU assignment.

Virtual FU assignment is a greedy mapping of operations to FUs. The mapping process does not take any data dependence information into account. To start the process, a cheapest FU is identified for each operation to calculate its inherent cost. This corresponds to the least expensive FU in the library to implement the operation if it were considered in isolation. Operations are then sorted from highest to lowest inherent cost.

The first operation in the sorted list that has not been assigned a virtual FU is the seed. All FUs in the library that could implement this operation are considered as candidates. Then, assignment of unbound operations to each of the candidate FUs is considered. The list is considered from the seed downward (most expensive to least expensive) where unbound operations that are compatible with the candidate FU are assigned one by one. Up to II operations may be assigned to a FU. To select the best choice among the candidate FUs, the *Desirability* of each is calculated as follows:

$$Desirability(VFU) = \sum_{op_i} \frac{CheapestFU(op_i)}{II} - VFUcost \tag{1}$$

The desirability measures how close the cost of the candidate FU is to the inherent cost of the assigned operations. The candidate FU that is closest (highest desirability) is chosen. The virtual FU assignment process continues until all operations have been assigned.

Available FUs: II = 2	Adder: 10 gates/bit Subtractor: 10 gates/bit Adder-Subtractor: 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	A: Adder, 32-bit, I1, I2, Des = -130 B: Adder-Subtractor, 32-bit, I1, I3, Des = -96
Seed: I2 Choose option A	A: Adder, 6-bit, I2, I4, Des = -5 B: Adder-Subtractor, 6-bit, I2, I4, Des = -35
After virtual FU assignment:	Adder-Subtractor: 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 3: Example application of width clustering.

The second step of width clustering is to perform clustering of the virtual FUs. Operations are indirectly clustered through their virtual FU assignment. For this step, a simple threshold is utilized. The virtual FUs are sorted from highest to lowest width. The cluster is initialized with the widest unbound FU. The ratio of the cluster width to each unbound FU is calculated. When this ratio falls below some threshold (e.g., 2.0), the FU is added to the cluster. Then, a new seed is chosen and the process is repeated until all virtual FUs are assigned to a cluster.

To illustrate the application of width clustering, the example in Figure 3 is presented. The example consists of 4 operations, 3 adds and a subtract, and an II of 2 is used. The example FU library has three elements: adder, subtracter, adder-subtracter. The ops 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 desirability of: $(((320+60)/2)-320) = -130$. With option B, the highest cost operation that is compatible is I3, yielding a desirability of: $(((320+320)/2)-416) = -96$. Hence, option B is chosen. The next seed chosen is I2, and with a similar calculation, option A is chosen. After virtual FU assignment is complete, there are 2 FUs: a 32-bit adder-subtracter assigned operations I1 and I3; and a 6-bit adder assigned operations I2 and I4.

Virtual FU clustering is then performed. Assuming a cluster ratio of 2, each virtual FU is assigned its own cluster. Hence after width clustering is complete, there are 2 clusters, (I1,I3) and (I2,I4). Note that this example is too simple to show an interesting virtual FU clustering.

5 Ongoing Work

We are currently evaluating the designs produced by the PICO system using bitwidth information. Preliminary experiments show that gate counts improve significantly across a range of multimedia loop nests. On average, a factor of two reduction in gate count is observed. One interesting issue is identifying the application characteristics when bitwidth sensitive code generation is most important. As one could anticipate, bitwidth analysis provides larger improvements when more narrow data is utilized. The largest gains are when expensive operations, such as multiplies, are narrowed. The width clustering works best when for a given operation class, there is a range of bitwidths from wide to narrow. When such a range exists, the scheduler frequently makes mistakes by not mapping similar width operations to common hardware resources. Thus, more cost-effective designs are achieved with the more intelligent mapping afforded using width clustering.