High-Level Synthesis of Nonprogrammable Hardware Accelerators

Robert Schreiber* Shail Aditya B. Ramakrishna Rau Vinod Kathail
Scott Mahlke Santosh Abraham Greg Snider

Hewlett-Packard Company, Laboratories, Palo Alto, California 94304

Abstract

The PICO-N system automatically synthesizes embedded nonprogrammable accelerators to be used as co-processors for functions expressed as loop nests in C. The output is synthesizable VHDL that defines the accelerator at the register transfer level (RTL). The system generates a synchronous array of customized VLIW (very-long instruction word) processors, their controller, local memory, and interfaces. The system also modifies the user's application software to make use of the generated accelerator. The user indicates the throughput to be achieved by specifying the number of processors and their initiation interval. In experimental comparisons, PICO-N designs are slightly more costly than hand-designed accelerators with the same performance.

1. Introduction

An ever larger variety of embedded ASICs is being designed and deployed to satisfy the explosively growing demand for new electronic devices. Many of these devices handle demanding multi-media computations. In many such ASICs, specialized nonprogrammable hardware accelerators (NPAs) are used for parts of the application that would run too slowly if implemented in firmware on an embedded programmable processor. Rapid, low-cost design, low production cost, and high performance are all important in NPA design. In order to reduce the design time and design cost, automated design of NPAs from high-level specifications has become a hot topic.

One of the principal goals of the HP Labs *Program-In-Chip-Out* (PICO) project is to automate the design of NPAs. The PICO-N user identifies a performance bottleneck and isolates it in the form of a loop nest. The source code for this key loop nest is used as a behavioral specification of an NPA. To obtain significant performance on this nest, PICO-N generates a highly parallel, special-purpose systolic array. The system produces the RTL design for the systolic array, its control logic, its interface to memory, and its control and data interface to a host processor. The design is expressed in synthesizable, structural VHDL. The original source is rewritten to make use of the hardware accelerator.

The current version of PICO-N has some limits on the acceptable input. The loop bounds must be constant and the array references, except for references to lookup tables, must be affine. The data-flow dependences between iterations must be uniform – they must have constant distances independent of the size of the iteration space.

Automatic synthesis of efficient general-purpose, programmable VLIW architectures is another aspect of the PICO research program that has been reported previously [13].

*Corresponding Author. `schreiber@hpl.hp.com`

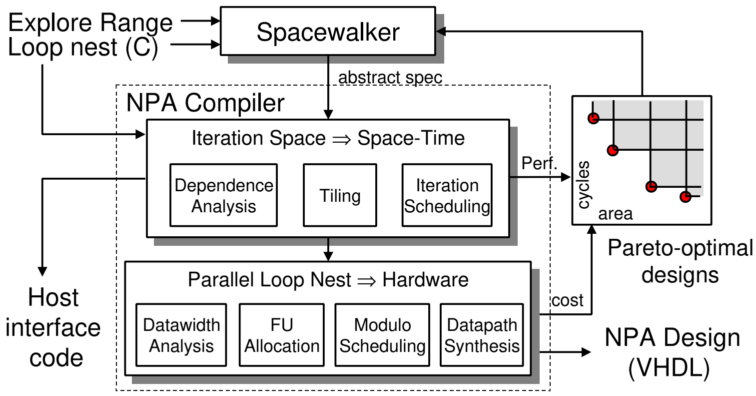

Figure 1. NPA Design System Components

PICO-N fully automates the design task for NPA realizations of affine loop nests with uniform flow dependences. The designs generated are competitive with hand-made designs. We believe it is the first such system that is complete, fully automated, and effective in creating cost-competitive architectures. The remainder of this paper describes the NPAs that PICO-N generates and its methods of generating them, and gives experimental results.

2. Overview of the PICO-N System

The components of the PICO-N system are shown in Figure 1. The user provides the loop nest in C as well as a range of architectures to be explored. A **spacewalker** is responsible for finding the best (*i.e.,* the Pareto optimal[1]) NPA architectures in this design space. 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 an RTL architecture via a compilation 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, as specified in the abstract architecture specification — the parallel loops are loops 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. An operation-level analysis and optimization phase in which classical optimizations and word-width minimization are done to the loop body;
5. A processor synthesis phase in which the processor's computation assets are allocated, the op-

[1]A design is Pareto optimal if no explored design has both lower estimated cost and better estimated performance.

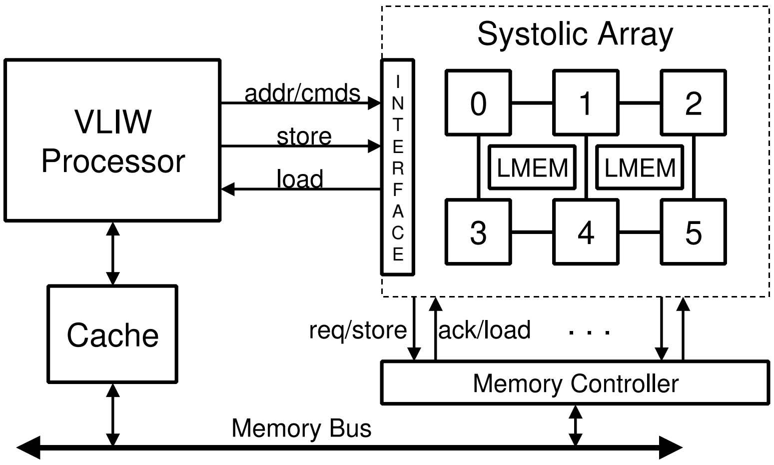

Figure 2. PICO's Generic NPA Architecture

erations and the data of the loop body are bound to these assets and scheduled relative to the start time of the loop (software pipelining), and the processor storage assets and intra- and inter-processor interconnects are created;

6. A system synthesis phase in which multiple copies of the processor are allocated and interconnected, and the controller and the data interfaces are designed;
7. A VHDL output and cost estimation phase.

The parallel form of the scheduled, mapped, register promoted, recurrenced loop nest is regenerated as C after Step 3, for examination and possibly modification by the user and for verification at the C level.

PICO-N synthesizes all the needed components of the NPA: the processor array, the array controller, local memories, an interface to global memory, and a control and data interface to the host. Both intra- and inter-processor interconnects are fully specified. Figure 2 illustrates the generic NPA architecture, which is discussed further in Section 4.

3. From Iteration Space to Space-Time

Input: Loop Nest and Required Throughput. The user provides a perfect loop nest in a subset of C. (A nest is perfect if there is no code other than a single embedded `for` loop in the body of any but the innermost `for` loop.) The language restrictions enable an exact dependence analysis and efficient hardware implementation. Salient restrictions include: no pointer dereference, affine array indices (except for *read-only* arrays that occur only on the right-hand side in the loop body), and constant loop bounds. If-then-else constructs are allowed and are removed by if-conversion. For violation of some of these rules, the current version of the system gives understandable error messages; for others it misbehaves. But all the rules are in principle capable of being checked and enforced by the compiler.

In addition, the spacewalker specifies parameters of the NPA that cannot be inferred from the code. These are: the number of processors in the NPA; their arrangement as a one or two dimensional mesh; the throughput of each processor in the form of the initiation interval (II); and the global memory bandwidth allocated to the NPA. The II is the number of clock cycles between the start times of successive loop iterations. The system designer may give the spacewalker an allowable range for each of the independent parameters, thereby delimiting the design space to be explored.

The source language has some extensions, in the form of pragmas. These allow the user to declare nonstandard data widths; to indicate that certain global variables are not live-in or not live-out (so that we do not have to depend on whole-program analysis); and to advise PICO-N to create local memory for certain arrays (lookup tables, for example).

Output: Synchronous Parallel Nest. The goal of the loop-transformation phase is a synchronous parallel nest, semantically equivalent to the original, that fully and explicitly defines the behavior of the NPA at the level of whole loop iterations and processors. It plays this role by virtue of the following characteristics: (i) The outer loop is a loop over clock cycles at which the inner loop is scheduled; (ii) The inner loops are parallel loops over the physical processor axes; (iii) Registers within the accelerator are expressed in the form of arrays indexed by clock cycle and processor. References to these make temporal delay and interprocessor communication explicit; for example:

```
X[t][p1][p2] = X[t - 5][p1][p2 + 1]
```

is a communication of the value of X computed five cycles earlier on a neighboring processor. These spatio-temporally indexed arrays, referred to as extended virtual registers (EVRs) [10], are realized as synchronous register FIFOs in the generated processors; (iv) References to main memory (for live-in and live-out data) are indicated by dereference of global pointers.

The processor synthesis phase (number 5 in the list above) derives a more detailed machine description, at the level of individual operations, machine cycles, and function-units.

Dependence Analysis. In the first compilation phase, a full value-based analysis of true dependence is done. The loop nest is first put into static single assignment form. For every right-hand side occurrence of an array element, the left-hand side occurrence that writes this element is identified, and the difference between the reader's and the writer's loop index vectors is the distance vector of this true dependence. We require that all true dependences have constant inter-iteration distance vectors.

Dependence analysis is simplified by the restriction that we make on the form of the affine references to an array. All references to an array that occurs on the left-hand side must have an affine map from loop iteration coordinates to array indices, and all references to a given such array must share the same linear form. Only the constants are allowed to differ. Thus, `a[i][j]` and `a[i+1][j-1]` may coexist, but `a[i][j]` and `a[i][0]` or `a[j][i]` cannot. We have found it possible to enforce this rule on a wide variety of interesting embedded loop nests (but not, however, on FFT).

An occurrence on the right-hand side can alias with one on the left-hand side and a dependence can exist only if the difference in the constant vectors of their index maps is in the lattice generated by the linear part of the affine index map. For example, `a[i+j][i-j]` cannot alias with `a[i+j][i-j+1]` because the difference in the constants in these affine maps (the vector $(0, -1)$) is not in the lattice generated by the linear form (the matrix $\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$). We say that two occurrences are *alias equivalent* if their constants differ by a vector in the generated lattice. This is an equivalence relation on occurrences of an array, and it is easy to check. We allocate one EVR for each equivalence class present in the program. This method generalizes recent work on load elimination by Weinhardt and Luk [16].

We also perform an analysis of input dependences for read-only arrays: we find a set of integer vectors that generates the lattice of iterations that read the same array element (the lattice of integer null vectors of the linear form of the affine index map). We use this basis to generate code that pipelines array elements (using an EVR per equivalence class of references) through the set of loop iterations that read them, performing loads of these values from memory only at iterations on the edge of the iteration space.

Scaling the Memory Wall. Limited memory bandwidth severely restricts the performance of most architectures, especially for data-intensive kernels. The NPAs synthesized by PICO-N overcome the memory wall by a combination of local memories and optimal register promotion. The user may annotate an array with a pragma directing PICO-N to keep that array in a dedicated local memory. In addition, no global-memory-resident datum generated in the loop nest is ever written to and then later re-read from global memory: it resides in a register throughout its lifetime. No live-in datum is ever read more than once from global memory. No datum is written to and later overwritten in global memory. Global memory traffic is therefore minimized. PICO-N manages this by using full dependence information to identify and eliminate all unnecessary memory traffic, and by allocating a sufficient set of processor registers to hold all live values without spill.

Clearly, eliminating inessential memory traffic through register promotion costs hardware for registers. We minimize this cost by tiling the iteration space. (Tiling was introduced in systolic computation by Moldovan and Fortes [7].) Tiling adds additional outer loops to a nest. PICO-N generates an accelerator for the inner loop nest over the iterations within a tile. The outer loops over tiles run sequentially on a host. The key issue is the rectangular tile shape $t = [t_i]$ (a vector giving the n extents of the tile.) PICO-N uses the array references that occur in the loop body to calculate the memory traffic $M(t)$ per tile. From the tile volume $K(t) \equiv \prod t_i$ and the specified throughput (the processor count (P), and initiation interval I) it calculates the processing-based bound $T(t) \equiv (K(t)I/P)$ on execution time for the tile. It then estimates the memory bandwidth needed to sustain this performance as $U(t) \equiv (M(t)/T(t))$. PICO-N uses constrained combinatorial optimization to minimize the tile volume subject to the constraint $U(t) \leq B$ where B is the memory bandwidth allocated to the NPA. Of course, it must choose a tiling that conforms to the Irigoin-Triolet conditions on the tile shape and the dependence distance vectors [5]. There may therefore be dimensions in which the tile extent must be the full iteration space extent.

Iteration Mapping and Scheduling. Iterations are the primary objects that we map and schedule. For an n-deep nest we identify each iteration with an integer n-vector $\vec{j}$, the vector of its loop indices. The schedule is a shifted-linear schedule: iteration $\vec{j}$ will be started at time $\tau \cdot \vec{j}$ where τ is an integer vector that we must choose, and $\cdot$ represents dot product. Let $\mathcal{O}$ be the set of operations found in the loop body. Then the instance of an operation $o \in \mathcal{O}$ at iteration $\vec{j}$ will be started at time $\tau \cdot \vec{j} + \rho_o$, where the scalars ρ will later be determined by a modulo scheduler.

The mapping from iterations to processors is done by orthogonal projection and clustering. We choose to map iterations first to an $(n-1)$-dimensional array of virtual processors by projecting out one of the iteration space dimensions. Next, we assign a *cluster* (a rectangular subset) of virtual processors to each physical processor. The projections we allow are parallel to the original axes of the iteration space. We allow the spacewalker to try all n projection directions, and use Pareto filtering to find the good choices. (We do not allow oblique projections: so far, we have not seen examples in which this limitation leads to serious inefficiency.)

We have discussed iteration scheduling in detail in another report [2]. We summarize the basic points here:

- The iteration schedule (τ) can be determined first, without knowledge of the operation schedule (*i.e.,* of ρ).
- Loop-carried dependences constrain the iteration schedule: τ must lie in a polyhedron $A\tau \geq b$ where A is determined by dependence analysis and b depends on the latencies of the available hardware function units. Any schedule τ in this polyhedron schedules dependent iterations to

begin after their dependence predecessors.

- From among the legal iteration schedules, we must choose one that is consistent with the clustering of virtual processors to physical processors. In particular, we want the schedule to assign one iteration to each physical processor every II cycles. We call such a schedule a *tight* schedule.
- We have developed an efficient algorithm for enumerating *all* of the tight schedules for a given cluster shape [2]. Using this technique, we are able to quickly find legal, tight schedules. We then choose the schedule from among these by applying a goodness criterion that measures total schedule length and an estimate of the gate count of the accelerator as a function of the schedule.

  Let us be a bit more precise. We have found a formula for tight schedules that is necessary and sufficient. For example, consider a three-deep loop nest, with iterations indexed $\vec{j} = (j_1, j_2, j_3)$ and with loop bounds $0 \leq j_i < 20, \quad i = 1, 2, 3$. Let iteration $\vec{j}$ be mapped to virtual processor (j_1, j_2). Cluster the 20×20 array of virtual processors on a 4×2 array of physical processors with a cluster shape of $C = (C_1, C_2) \equiv (5, 10)$. Then the schedule

  $$\tau = (\eta_1, C_1\eta_2, C_1C_2\eta_3)$$

  is tight if η_i and C_i are relatively prime for $i = 1$ and 2, and $\eta_3 = \pm 1$. This formula yields all the tight schedules if we allow first a permutation of the cluster axes; in other words, we include as well the schedules

  $$\tau = (C_2\eta_1, \eta_2, C_1C_2\eta_3) \quad .$$

Recurrence-Based Control. In its final form, the transformed parallel loop body contains generated code that we call *housekeeping* code. On a given processor at a given clock, this code computes the coordinates (the loop indices) of the iteration being started, the addresses to be used in global memory accesses, the position of the active virtual processor within the cluster, and some comparisons to determine if a datum is to be read or written to global memory or communicated from a neighboring processor. Iteration coordinates are a well-defined function of the processor and time coordinates, and all the other quantities mentioned can in turn be computed from the iteration coordinates. But a straightforward computation in this manner is very expensive and can often dwarf the cost of the original loop body. This problem is well known in loop parallelization.

We have reduced the cost of computing coordinates, addresses, and predicates by using a temporal recurrence mechanism to update their values rather than compute them from first principles. Their values for the first few clock cycles (usually three cycles) are live-in and must be downloaded. At each clock, the update requires just one addition for each coordinate and memory address of interest. In addition, one comparison must be performed for each axis in which the cluster shape is greater than one. The values of almost all predicates are periodic with period equal to the cluster size γ, so they may be obtained by downloading the first γ of their values into a ring buffer [2]. In particular, an affine inequality in the iteration space coordinates that does not involve the mapped out dimension will be periodic.

Chen [1] and Rajopadhye [9] also proposed propagation rather than recomputation as a means of computing predicates (which they call *control signals*.) They do not address the computation of coordinates and addresses. They look at predicates generated by linear inequalities in the original iteration coordinates, which can be transformed into equivalent linear inequalities in the time and virtual processor coordinates. Such predicates remain constant in a hyperplane of virtual space-time, and so they can be computed at the boundary (which may be at the start, or at a boundary processor) and propagated through the interior, in much the same way that we treat affine, read-only input data. Our periodicity approach for predicates is quite different, and is not applicable to as large a class as theirs. For periodic predicates, it does all computation in the host processor (before the start of the systolic computation) and so it requires not comparison hardware in the array at all. It does no spatial propagation. On the other hand, it requires a fixed storage cost of γ bits for each prediate. As an additional optimization, we probably should use the scheme of Chen or Rajopadhye for nonperiodic affine predicates.

4. From Parallel Program to Hardware

The iteration-level transformations bring the loop nest to a form ready for hardware synthesis. In the next steps, the computation within each transformed iteration is mapped and scheduled to a single non-programmable processor with the desired II. Multiple instances of such processors are then interconnected into a systolic array. The various steps of this process are briefly described below.

Word Width. The first step towards hardware realization is to minimize the width of each operator and operand. We allow user annotation of the widths of variables via pragmas. Compiler-generated variables are similarly sized according to the range of values they must hold. We use this information to infer width requirements for all program data, both named variables and compiler-generated temporaries, and all operations. We may discover that the user has been overly conservative, and that some variables may not need all of the bits that the user has allowed. The analysis derives reduced widths for each program variable, temporary variable, and operator within the application. We describe our method of width inference and the cost reduction we derive from it in another report [6]. [2]

Initial Function Unit Allocation. This step selects a least-cost set of function units (FUs) that together provide enough throughput for each opcode occurring in the transformed loop iteration. This problem is formulated as a mixed integer linear programming (MILP) problem as follows.

The FUs are drawn from a macrocell library and are classified into several types. Each FU type has a per-unit cost represented by the real cost vector $c = [c_f]$, and it has a repertoire of operation types (i.e. opcodes) that it can handle. The boolean variable r_{of} is true if the FU type f can handle operation type o. We assume all FUs are pipelined and can accept one new operation every cycle;[3] the operation latency need not be one.

The transformed loop body is characterized by a count vector $k = [k_o]$ giving the number of operations of each type.

The goal is to allocate zero or more FUs of each type, expressed as an integral allocation vector $n = [n_f]$. We would like to minimize the price $c \cdot n$ paid for these. But we must require that the cycles available on the allocated FUs can be distributed to cover the operations that occur in the loop body. We introduce nonnegative auxiliary variables $x = [x_{of}]$ that represent the number of operations of type o assigned to FUs of type f. This leads to the following MILP:

$$\text{minimize} \quad c \cdot n \quad \text{subject to}$$

$$\begin{aligned} \sum_o x_{of} &\leq n_f I \\ \sum_f x_{of} r_{of} &\geq k_o \end{aligned}$$

where I is the initiation interval.

[2] Very similar analyses and optimizations have recently been reported by Stephenson, Babb, and Amarasinghe [14].

[3] This assumption is easily relaxed to include block-pipelined units that may accept a new operation every n cycles.

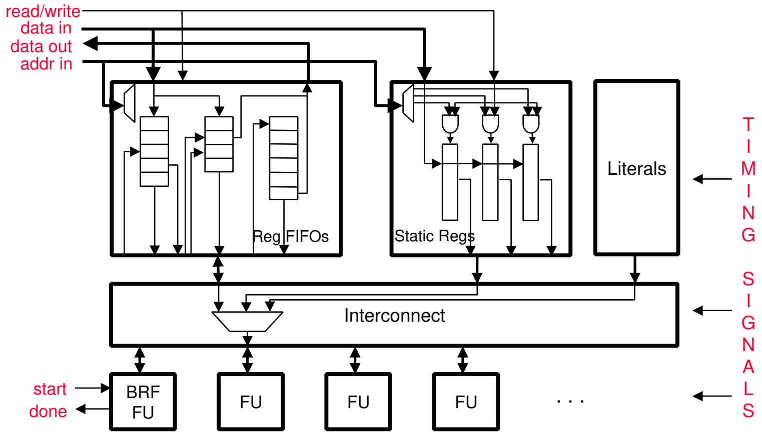

Figure 3. The processor datapath

Each of the first constraints bounds the number of cycles used (for all operations) on an FU type by the number available. The second set of constraints guarantees that the required throughput (for each opcode type) can be achieved by the contributions from the allocated FUs.

We have found that it is easier to solve this MILP when x is allowed to be real; the FU allocation is essentially as good as when x as well as n is constrained to be integer. This formulation is small and inexpensive to solve, since its size depends only on the number of opcode and FU types and not on the size of the loop nest. For example, CPLEX takes about 16 seconds on a Pentium PC to solve a problem with 40 FU types and 30 operation types.

Operation Mapping and Scheduling. The next step is to extract a compiler-oriented view of the allocated hardware in the form of a machine-description [11]. Even though the final hardware is supposed to be non-programmable, at this stage we assume a fully interconnected abstract architecture which is completely programmable. This allows our re-targetable VLIW compiler, Elcor, to be used to schedule and map the loop iteration onto the allocated function units. Once the scheduling and mapping of operations to FUs is done, only the required datapath and interconnect among the FUs is actually materialized and the rest is pruned away leaving a minimal, non-programmable architecture.

The VLIW compiler is used to perform modulo-scheduling [10] of the transformed iteration at the desired II using various heuristics. The width heuristic tries to reduce hardware cost by mapping similar width operators to the same FU. The interconnect heuristic tries to reduce the fan-out and fan-in of the FUs by co-locating operations that have co-located successors or predecessors. The register heuristic attempts to share the same register file for multiple data items. The auxiliary variables (x) in the MILP formulation of FU allocation may also be used to heuristically guide operation mapping. This is an area of active investigation.

It is possible for the modulo scheduler to fail at the desired II with the initial FU allocation. In traditional compilation the target hardware is fixed, and the fallback is to increase the II and retry. In our setting, the right thing to do is a reallocation, in which more FU hardware is added before a retry to schedule at the desired II.

Datapath Generation. The processor datapath schema is shown in Figure 3. The various operands of the loop body are materialized using one synchronous register FIFO per EVR (extended virtual register) in the loop body, one physical register per static virtual register, and hardwired literals. When the II is greater than 1, multiple opcodes map to each of the FUs, and multiple sets of

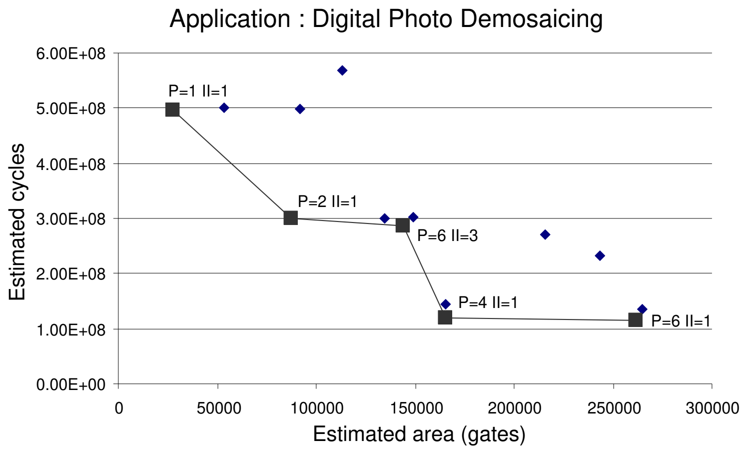

Figure 4. Example Pareto Set

operands are routed to the FU input ports. The FU selects one out of II sets of opcode and operands at each cycle using multiplexors.

The various EVR operands are read from and written to the appropriate register slot in the FIFO according to the schedule time of the corresponding operation. This gives rise to a sparse set of read/write ports to the various FIFOs.

Iteration Control. The schema shown above directly implements the pipeline schedule produced by the compiler, which uses predicated operations to fill and drain the pipeline automatically. The loop hardware is initialized for execution of a tile of data by setting loop trip count and live-in register values within the FIFO and static registers. The setting of a start flag signifies the start of the loop. A counter is used to count up to II, at the end of which the loop count is decremented and all register FIFOs shift down by one slot to make room for the operands of the next iteration, and the cycle repeats. Once the desired number of iterations has been initiated, the pipeline is drained; it then sets a done flag. This flag can be queried from the host processor to recognize that the execution is complete and any live-out values may be uploaded.

System Synthesis. The complete NPA system consists of multiple instances of the processor, configured as an array with explicit communication links as shown in Figure 2. The processors may share array-level local memory via dedicated local memory ports. Since global memory bandwidth is constrained externally, the processors communicate with the global memory through a controller which consists of an arbitrator and request/reply buffering logic. The entire NPA array, including the processor registers and array-level local memories, behaves like a local memory unit to the host processor. It may be initialized and examined using this interface. Iteration control and data exchange commands are also encoded as local memory transactions across this interface.

5 Experimental Evaluation

We have used our system to compile over thirty C loop nests to hardware. The loops are drawn from printing, digital photography, genomics, image compression, telephony, and scientific computing domains. Two examples illustrate some typical experiments performed and the data collected. In the first, a design-space exploration is performed on a digital photography demosaicing application. In this experiment, the number of processors is varied from 1 to 6 and the II of each processor

Components	II = 1	II = 2	II = 4
branch	1	1	1
multiply	1	1	1
move	23	6	3
add	19	9	4
sub	3	2	0
addsub	0	0	1
shift	1	1	1
compare	14	4	2
and	3	0	0
ld_st_global	8	4	2
ld_st_local	7	4	2
register	259	135	96
Gates	II = 1	II = 2	II = 4
function unit	19025	12962	9550
register	15590	13320	9970
register select	1248	893	660
multiplexor	0	3916	4375
misc. logic	806	479	425
total	36669	31571	24980

Table 1. Effect of II on Cost Breakdown

from 1 to 4. Figure 4 shows the resulting cost/performance tradeoff with the Pareto points marked by their configuration. The II=1 designs are particularly effective for this application.

The second experiment examines in more depth the effect of increasing II on cost for an image sharpening application. Table 1 presents cost breakdowns for three single-processor designs: II=1, II=2, and II=4. (The throughput obtained by an HP PA-8000 class RISC on this application is equivalent to an II of 175. Thus, one processor with a small II is an interesting design point.) The top portion of the table presents the number of function unit and register components used for each design. The bottom portion of the table shows the gate count breakdowns for each design. From II=1 to II=2, a modest cost reduction is obtained by reducing the function unit and register cost by large amounts. A side effect of increasing II is that more multiplexors are needed to gate data into shared resources, but their cost is outweighed by the savings. The cost is further reduced when II is increased to 4, but again the reduction is by much less than a factor of two. Note that an expensive multiplier is allocated because of a single multiply operation, regardless of II. If slower, less expensive multipliers were available in the macrocell library, these would be chosen at the higher IIs.

To date, we have calibrated our automatic designs with hand generated designs for two applications. The manually designed accelerators were included in ASICs embedded in HP printers. In both cases, for the same level of performance, our gate counts are within 10% of the hand designs.

The compile time for a single design-space point currently ranges from about 90 seconds to about 10 minutes, depending on the complexity of the loop body and the depth of the nest.

6. Related Work and Conclusions

The basic schema for systolic synthesis described by Quinton and Robert [8] underlies the architecture of PICO-N. The IRISA system [17], is a comparable system for systolic synthesis, taking systems of affine recurrence equations written in the ALPHA language and generating a hardware description. The HIFI [4] system from Delft accepts loop nest programs as we do, and does value-based dependence analysis as well. Other systems for the automated design of embedded multiprocessors include CATHEDRAL-II [3].

There are also some practical limitations to the working version of PICO-N. Most notably, it requires that the flow and output dependences in its loop nest be uniform; other systolic synthesis systems allow a system of affine recurrence equations as input. Techniques for conversion of affine recurrence equations to uniform recurrence equations have been developed by several researchers [15, 12]. PICO-N currently restricts loop bounds to be constant. It requires the user to express his computation as a perfect nest.

While earlier systems may have individually incorporated some of its features, PICO-N goes beyond previous efforts directed at NPA compilation in fully integrating the following characteristics:

- PICO-N exploits loop-level and instruction-level parallelism; it performs both loop parallelization and efficient pipelined datapath synthesis;
- Using advanced techniques for dependence analysis, PICO-N accepts restricted C code rather than a system of uniform recurrence equations as input;
- PICO-N exploits both tiling for reducing required memory bandwidth and clustered mapping of virtual to physical processors to map a computation to a fixed-size, fixed-throughput array;
- PICO-N uses advanced techniques for load-store elimination and pipelining of input data;
- PICO-N implements a new, practical method for systolic array scheduling;
- PICO-N uses a new, efficient method of code generation for parallel loops;
- PICO-N allows the user to extend C with the use of pragmas for data width specification and for locating arrays in local memory;
- PICO-N uses new techniques for data width inference to minimize datapath cost;
- PICO-N uses a new resource allocation technique for low-cost, special-purpose datapath synthesis;
- PICO-N uses new variations of modulo scheduling for hardware-cost-sensitive software pipelining.

Acknowledgements. Bruce Childers, Andrea Olgiati, Rodric Rabbah, Tim Sherwood, and Frédéric Vivien, have all helped with the design and implementation of parts of the PICO-N software. We also thank Alain Darte and Lothar Thiele for their important contributions.

Appendix. The following example illustrates several steps in the compilation process.

```
/* Original Nest */
for (i1=0; i1<8192-16; i1++) {
  y[i1] = 0;
  for (i2=0; i2<16; i2++)
    y[i1] = y[i1] + w[i2]*x[i1+i2];
  }


/* Perfect Nest */
for (i1=0; i1<8192-16; i1++)
  for (i2=0; i2<16; i2++)
    y[i1] = y[i1] + w[i2]*x[i1+i2];


/* After Tiling */
for (i2T = 0; i2T <= 15; i2T += 4)
  for (i1 = 0; i1 <= 8175; i1++)
    for (i2P = 0; i2P <= 3; i2P++)
       y[i1] = y[i1] +
          w[i2T + i2P] * x[i1 + i2T + i2P];
```

```
/* After Load-Store Elimination */
for (i1 = 0; i1 <= 8175; i1++)
  for (i2P = 0; i2P <= 3; i2P++)
    {
      i2 = i2T + i2P;
      EVR_w[i1][i2P] = EVR_w[i1 - 1][i2P];
      if (1 <= i2P)
        EVR_tmp4[i1][i2P] = EVR_y[i1][i2P - 1];
      else
        EVR_tmp4[i1][i2P] = y[i1];
      if (i1 <= 8174 \& 1 <= i2P)
        EVR_x[i1][i2P] = EVR_x[i1 + 1][i2P - 1];
      else
        EVR_x[i1][i2P] = x[i1 + i2];
      EVR_y[i1][i2P] = EVR_tmp4[i1][i2P] +
        EVR_w[i1][i2P] * EVR_x[i1][i2P];
      if (i2P == 3)
        y[i1] = EVR_y[i1][i2P];
    }
```

```
/* In Optimized Space-Time Form */
for (t = 0; t <= 8184; t++)
  for (p = 0; p <= 3; p++)
    {
      i1 = 1 + EVR_i1[t - 1][p];
      addr_y = EVR_addr_y[t - 1][p] + 2;
      addr_x = EVR_addr_x[t - 1][p] + 1;
      unif_p0 = EVR_i1_unif_p0f[t - 1][p];
      unif_p = EVR_i1_unif_pf[t - 1][p];
      EVR_w[t][p] = EVR_w[t - 1][p];
      if (0 <= i1 \& i1 < 8176)
        {
          if (unif_p0)
              EVR_tmp4[t][p] = EVR_y[t - 3][p - 1];
          else
              EVR_tmp4[t][p] = *addr_y;
          if (i1 <= 8174 \& unif_p0)
              EVR_x[t][p] = EVR_x[t - 2][p - 1];
          else
              EVR_x[t][p] = *addr_x;
          EVR_y[t][p] = EVR_tmp4[t][p] +
                        EVR_w[t][p] * EVR_x[t][p];
          if (unif_p)
              *addr_y = EVR_y[t][p];
        }
      EVR_i1_unif_p0f[t][p] = unif_p0;
      EVR_i1_unif_pf[t][p] = unif_p;
      EVR_addr_x[t][p] = addr_x;
      EVR_addr_y[t][p] = addr_y;
      EVR_vp[t][p] = vp;
      EVR_i1[t][p] = i1;
    }
```

References

[1] M. C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, pages 461–491, 1986.

[2] A. Darte, R. Schreiber, B. R. Rau, and F. Vivien. A constructive solution to the juggling problem in systolic array synthesis. In *Proceedings, International Parallel and Distributed Processing Symposium*, Cancun, Mexico, May 2000.

[3] H. De Man, J. Rabaey, P. Six, and L. Claesen. CATHEDRAL-II: a silicon compiler for digital signal processing multiprocessor vlsi systems. *Design & Test of Computers*, pages 13–25, 1986.

[4] P. Held, P. Dewilde, E. Deprettere, and P. Wielage. HIFI: From parallel algorithm to fixed-size VLSI processor array. In *Application-driven architecture synthesis*, pages 71–94. Kluwer Academic Publishers, 1993.

[5] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the Fifteenth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 319–329, 1988.

[6] S. Mahlke, R. Schreiber, S. Abraham, and T. Sherwood. Wordlength inference in C code for ASIC synthesis. Technical Report in preparation, HP Laboratories, 2000.

[7] D. Moldovan and J. Fortes. Partitioning and mapping of algorithms into fixed size systolic arrays. *IEEE Transactions on Computers*, 35:1–12, 1986.

[8] P. Quinton and Y. Robert. *Systolic Algorithms and Architectures*. Prentice Hall International (UK) Ltd., Hemel Hempstead, England, 1991.

[9] S. V. Rajopadhye. Synthesizing systolic arrays with control signals from recurrence equations. *Distributed Computing*, 3:88–105, 1989.

[10] B. R. Rau. Iterative modulo scheduling. *International Journal of Parallel Processing*, 24:3–64, 1996. Also available as HP Labs Tech. Report HPL-94-115.

[11] B. R. Rau, V. Kathail, and S. Aditya. Machine-description driven compilers for EPIC and VLIW processors. *Design Automation for Embedded Systems*, 4:71–118, 1999.

[12] V. Roychowdhury, L. Thiele, S. Rao, and T. Kailath. On the localization of algorithms for VLSI processor arrays. In *IEEE Workshop on VLSI Signal Processing*. IEEE, 1989.

[13] V. K. Shail Aditya, B. Ramakrishna Rau. Automatic architecture synthesis of VLIW and EPIC processors. In *Proceedings of the 12th International Symposium on System Synthesis, San Jose, California*, pages 107–113, November 1999.

[14] M. Stephenson, J. Babb, and S. Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation PLDI*. ACM Press, June 2000.

[15] V. van Dongen and P. Quinton. Uniformization of linear recurrence equations: A step towards the automatic synthesis of systolic arrays. In *Proceedings of the International Conference on Systolic Arrays*, pages 473–481, San Diego, California, 1988. IEEE Computer Society Press.

[16] M. Weinhardt and W. Luk. Memory access optimization and RAM inference for pipeline vectorization. In *Field Programmable Logic and Applications, Proceedings of the 9th International Workshop, FPL '99*, volume 1673 of *Lecture Notes in Computer Science*, pages 61–70, New York, 1999. Springer-Verlag.

[17] D. Wilde and O. Sie. Regular array synthesis using ALPHA. In *Proceedings, Application-specific Systems, Architectures and Processors Conference*, San Francisco, June 1994. IEEE.