PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators *

Robert Schreiber† Shail Aditya Scott Mahlke Vinod Kathail
B. Ramakrishna Rau Darren Cronquist Mukund Sivaraman

Hewlett-Packard Laboratories, Palo Alto, California 94304-1126

Abstract

The PICO-NPA system automatically synthesizes nonprogrammable accelerators (NPAs) to be used as co-processors for functions expressed as loop nests in C. The NPAs it generates consist of a synchronous array of one or more customized processor datapaths, their controller, local memory, and interfaces. The user, or a design space exploration tool that is a part of the full PICO system, identifies within the application a loop nest to be implemented as an NPA, and indicates the performance required of the NPA by specifying the number of processors and the number of machine cycles that each processor uses per iteration of the inner loop. PICO-NPA emits synthesizable HDL that defines the accelerator at the register transfer level (RTL). The system also modifies the user's application software to make use of the generated accelerator.

The main objective of PICO-NPA is to reduce design cost and time, without significantly reducing design quality. Design of an NPA and its support software typically requires one or two weeks using PICO-NPA which is a many-fold improvement over the industry norm. In addition, PICO-NPA can readily generate a wide-range of implementations with scalable performance from a single specification. In experimental comparison of NPAs of equivalent throughput, PICO-NPA designs are slightly more costly than hand-designed accelerators.

Logic synthesis and place-and-route have been performed successfully on PICO-NPA designs, which have achieved high clock rates.

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 multi-media computations requiring high throughput. 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 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.

The Program-In, Chip-Out (PICO) project is a long-range research effort at HP labs, aimed at automating the design of customized, optimized, general and special-purpose processors. One aspect of PICO is the PICO-NPA system that automatically synthesizes nonprogrammable accelerators (NPAs) to be used as co-processors for functions expressed as loop nests in C. The PICO-NPA user

* Journal of VLSI Signal Processing, *to appear.*
† Corresponding Author. `schreiber@hpl.hp.com`

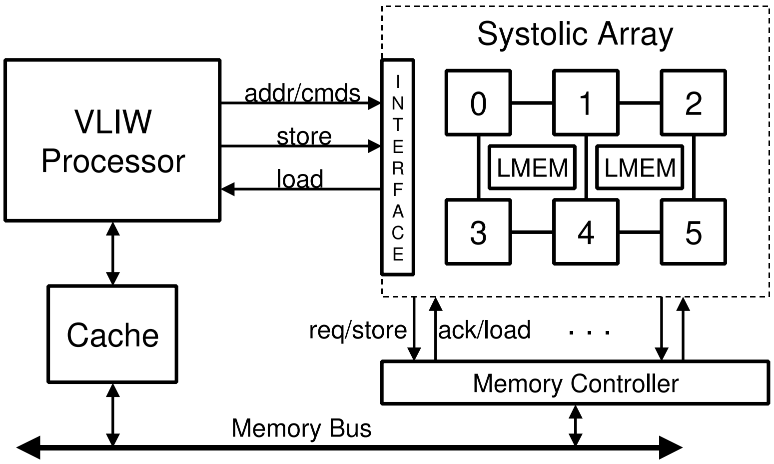

Figure 1. PICO's Generic NPA Architecture

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-NPA generates a special-purpose array of one or more datapath processors. 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 HDL. (The current implementation uses VHDL; a Verilog capability is in progress.) PICO-NPA rewrites the original source code to make use of the hardware accelerator. The nest is replaced by code that initializes the accelerator by downloading into registers any live-in scalar values, such as the memory addresses of input and output arrays, starts the accelerator, and waits for its completion.

Automatic synthesis of customized, general-purpose, programmable VLIW architectures, which may be assisted by one or more NPAs at the designers discretion, is another aspect of the PICO research program that has been reported previously [2].

Figure 1 illustrates the generic NPA architecture, which is discussed further in Section 4. PICO-NPA 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.

This paper describes the NPAs that PICO-NPA generates and its methods of generating them, and presents an extended example and some evaluations.

2 Overview of the PICO-NPA system

The components of the PICO-NPA system are shown in Figure 2. 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) NPA architectures in this design space [1]. 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 an efficient, detailed NPA design for the function performed by the nest that is consistent with the abstract architecture specification provided by the spacewalker. It expresses the design in the HDL. It also generates an accurate performance measurement and an estimated gate count for the NPA. The system designer gives

[1] A design is Pareto optimal with respect to a design space and a set of evaluation criteria (estimated cost, estimated power, estimated performance, for example) if no other design in the space is better in some respect and equally good in all others.

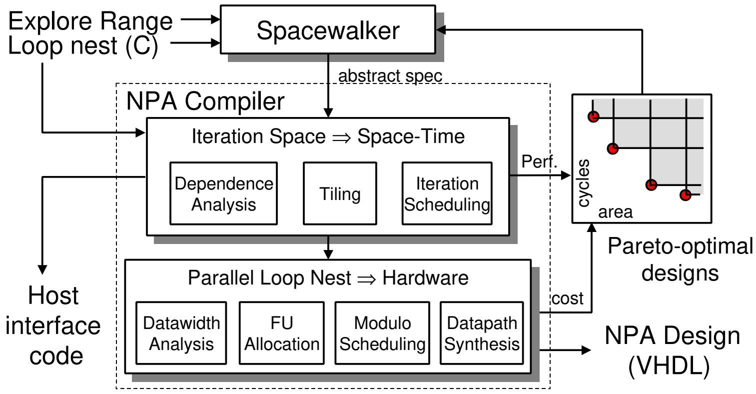

Figure 2. NPA Design System Components

the spacewalker an allowable range for each of the independent parameters, thereby delimiting the design space to be explored.

The NPA compiler transforms the loop nest into an RTL architecture via a sequence of phases:

1. An analysis phase in which array accesses and dependences are found;

2. A mapping/scheduling phase in which the mapping of iterations to processors and to clock cycles is determined;

3. A loop transformation phase that recasts the loop into an outer, sequential loop and a nest of inner, parallel loops. The outer loop iterates over the temporal extent of the schedule chosen in the previous phase, and the parallel loops iterate over the processor indices of the grid specified by the spacewalker. This form mirrors the hardware's structure, as specified in the abstract architecture specification. 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 dependence analysis, classical optimization, and word-width minimization are done to the transformed loop body;

5. A processor synthesis phase in which the processor's computation assets are allocated, the operations and the data of the optimized 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. An HDL output and cost estimation phase.

Phases 1–3 above are described in detail in Section 3. These steps are all implemented by the “front end”, an implementation based on the SUIF compiler infrastructure [22].

Phases 4–6 above are described in detail in Section 4. These steps are all implemented by the “back end”, an implementation based on the HP-Labs Elcor compiler, a retargetable, machine-description-driven VLIW compiler [16, 1].

3 The front end: transforming the nest to a parallel specification

The goal of the front end is to create a semantically equivalent nest that which describes exactly what array elements and registers are referenced and which operations are performed by each processor at each clock step.

The best way to illustrate this is through a detailed example. We will take a code for a FIR filter through PICO-NPA step by step, explaining each step along the way.

Here is the user's original loop nest:

```
/* Original Nest */
for (j1=0; j1<8192; j1++) {
  y[j1] = 0;
  for (j2=0; j2<16; j2++)
    y[j1] = y[j1] + w[j2]*x[j1+j2];
  }
```

3.1 Input to the synthesis system

3.1.1 Input loop nest

A loop nest is *perfect* if there is no code other than a single embedded "for" loop in the body of any but the innermost loop. The PICO-NPA user provides a perfect nest, written in C. If-then-else and switch-case constructs are allowed in the innermost loop body and are removed by if-conversion; goto is forbidden.

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-NPA to create local memory for certain arrays, for example, lookup tables.

In the example, there is code between the two loops: the nest is not perfect. But the initialization of `y` can be done in an earlier loop; so loop distribution can be used to make it perfect. For cases in which loop distribution is not possible, a nest like this one can be made perfect by sinking the imperfectly nested statements into the inner loop under a guard. At present, this analysis and code preparation is the user's responsibility, although it could be done with standard compiler technology.

After removal of the initialization to the host code, we have the perfect nest:

```
/* Perfect Nest */
for (j1=0; j1<8192; j1++)
  for (j2=0; j2<16; j2++)
    y[j1] = y[j1] + w[j2]*x[j1+j2];
```

3.1.2 Abstract architecture specification

The spacewalker specifies parameters of the implementation that cannot be inferred from the code.

The spacewalker first prescribes the number of processors. The processors that we generate are custom, parallel datapaths, which provide a pipelined implementation of the loop nest: several iterations may be active in the pipeline simultaneously. In a pipelined implementation of an inner loop on one processor, the *initiation interval* (II) is the number of clock cycles between the start times of successive loop iterations. The throughput is determined, to first order, by the achieved initiation interval. In PICO-NPA the spacewalker specifies the II to be achieved by the synthesized processors for the given loop nest.

For this example, we will assume that what is wanted is an array of two processors, each with an initiation interval of one.

The spacewalker also specifies a budget for the NPA's use of global memory bandwidth. For the example, we will use no more than two words per cycle of global memory bandwidth.

3.2 Tiling, mapping, schedule

The first job is a mathematical analysis to determine the tiling, mapping, and scheduling. This stage treats the iterations as indivisible tasks. It assigns each loop iteration, identified by its indices (in FIR: (j_1, j_2)), to one of the processors, and to a clock cycle. Furthermore, it decides on a tiling, which is a partitioning of the large, rectangular set of iterations (in FIR: $(0 \leq j_1 < 8192, 0 \leq j_2 < 16)$) into rectangular tiles. The NPA hardware will be designed to perform the work of one tile. The original loop nest will be replaced by a sequential, nest of loops over the set of tiles, in which the NPA is invoked to perform the work of one tile at a time.

3.2.1 Dependence analysis

Before the tile shape, the mapping, and the schedule can be chosen, we must find the dependence relations that constrain these choices. In FIR, dependence analysis reveals that the add operation at iteration (j_1, j_2) is directly dependent on the add at iteration $(j_1, j_2 - 1)$. We also analyze the uses of read-only data; we find that iterations (j_1, j_2) and (k_1, k_2) use the same element of the array `w` if and only if the difference in their loop index vectors is an integer multiple of the vector $(1, 0)$ (*i.e.,* if $j_2 = k_2$), and that for `x`, the axis along which elements are reused is $(1, -1)$. We also identify any antidependence and output dependence relations that constrain the mapping and scheduling of iterations to processors and clock cycles. PICO-NPA carries out its dependence analyses using the Omega system [12].

3.2.2 The memory wall: how we achieve performance with limited memory bandwidth

Limited memory bandwidth severely restricts the performance of most architectures, especially for data-intensive kernels. The NPAs synthesized by PICO-NPA overcome the memory wall by a combination of local memories and optimal register promotion for arrays with uniform dependences.

The user may annotate an array with a pragma directing PICO-NPA to keep that array in a local memory. This reduces demand for global memory bandwidth. It requires a user guarantee (in the form of the pragma) against aliasing through pointers.

The dependence relation involving `y` in FIR is *uniform*, meaning that the distance from an iteration to its dependence predecessor is a constant vector. In the load/store elimination step that is described later, PICO-NPA identifies and eliminates all unnecessary memory traffic for arrays that have uniform dependences. For such arrays, values that are generated in the loop nest are not written to and then later re-read from global memory: any such value resides in a register throughout its lifetime. To enable this optimization, PICO-NPA allocates enough processor registers to hold all live values without spill. Live-in data is not read more than once from global memory. No datum is written to and later overwritten in global memory. Global memory traffic is therefore minimized.

3.2.3 Why we tile the loop nest, and how we choose the tile shape

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 [11].)

Tiling adds additional outer loops to a nest. PICO-NPA generates an accelerator for the inner loop nest over the iterations within a tile. The outer loops over tiles run sequentially on a host.

Thus, tiling reduces the size of the iteration space that is implemented by the NPA. This reduces the demand for registers, for reasons that will become clear in Section 3.3.3.

In the FIR example with the given abstract architecture specification, PICO-NPA decided to break the iteration space into four tiles, each with shape $(8192, 4)$. Here is the tiled version of the FIR code:

```
/* Tiled Nest */
for (j2T = 0; j2T <= 3; j2T++)
  /* Inner Nest Over One Tile */
  for (j1 = 0; j1 <= 8191; j1++)
    for (j2 = 0; j2 <= 3; j2++)
      y[j1] = y[j1] + w[4 * j2T + j2] * x[j1 + 4 * j2T + j2];
```

The loop indices now traverse a space whose size is the tile shape. There is an outer loop over the tiles. In the j_1 dimension the tile covers the whole iteration space, so j_1^T is identically zero, and so it can be omitted. Within tile j_2^T, the iteration with indices (j_1, j_2) corresponds to iteration $(j_1, 4j_2^T + j_2)$ of the original iteration space. This means that there is a dependence from iteration $(j_1, 3)$ of tile j_2^T to iteration $(j_1, 0)$ of tile $j_2^T + 1$. So the tile loop must run in order of increasing j_2^T.

The tile shape in FIR was chosen by calculating that a tile of shape $(8192, 4)$ running on two processors with unit II has 32,768 iterations that are retired two on every cycle in the steady state. So the compute time will be about 16,384 cycles. Because of load/store elimination, we will load or store an array element only once; hence there will be 8192+3 words of x loaded, 8192 words of y loaded, 8192 words of y stored, and 4 words of w loaded, for a total of 24,583 memory references, which can be achieved with a bandwidth of two in the 16,384 available cycles. We could have defended a tile shape of $(8192, 3)$ on bandwidth grounds, but with two processors, the tile shape $(8192, 3)$ takes as much time per tile as $(8192, 4)$, and leads to six rather than four tiles. See Section 3.2.4

The key issue in tiling, then, is the rectangular tile shape $\vec{t} = [t_i]$, a vector giving the extents of the tile. PICO-NPA chooses it by minimizing the number of points in the tile, subject to the constraint of not requiring more global memory bandwidth than the spacewalker has budgeted. We have found this to be an easy discrete optimization problem to solve in practice.

In order to implement this strategy, PICO-NPA needs to be able to estimate the bandwidth required by a tile. It uses the affine references that occur in the loop body to calculate the memory traffic $M(\vec{t})$ per tile. Its method is derived from work of Gallivan, Jalby, and Gannon [6]. Non-affine array references, and affine references with nonuniform dependence relations, are not helped by load/store elimination, so PICO-NPA assumes that each such occurrence in the text of the loop body costs one memory access at each iteration.

As in the discussion of FIR above, PICO-NPA uses tile volume $K(\vec{t}) \equiv \prod t_i$, the processor count P, and initiation interval λ to calculate a processing-based estimate on execution time for the tile: $T(\vec{t}) \equiv (K(\vec{t})\lambda/P)$. It then estimates the memory bandwidth needed to sustain this performance as $U(\vec{t}) \equiv (M(\vec{t})/T(\vec{t}))$. The optimizer minimizes $K(\vec{t})$ subject to the constraint $U(\vec{t}) \leq B$, where B is the budgeted memory bandwidth.

Of course, the tiling must conform to the Irigoin-Triolet conditions for a legal tiling [8]. There may therefore be some dimensions that cannot be tiled; in these dimensions, the tile extent must be at least equal to the full iteration space extent.

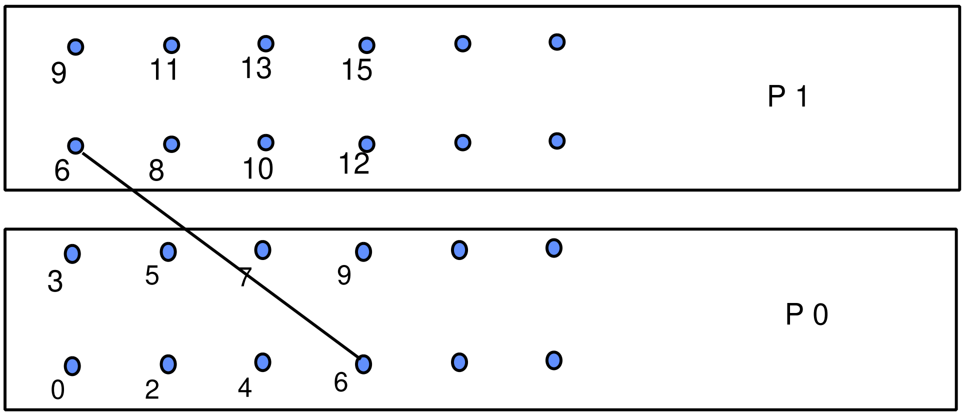

Figure 3. A tight schedule for FIR on two processors, $\vec{\tau} = (2, 3)$. The j_1 axis is horizontal. Dots represent iterations, and the numbers next to them are the times at which they begin. The oblique line is shows the wavefront of iterations scheduled a time $t = 6$.

3.2.4 Iteration mapping

Iterations are the primary objects that we map and schedule. In PICO-NPA the mapping from iterations to processors is done by orthogonal projection and clustering.

For the FIR example, the mapping to the two processors is chosen as follows. Iteration (j_1, j_2) is first conceptually mapped to a virtual processor (VP) by dropping the first component j_1. Thus there are four virtual processors, labeled $0 \le v = j_2 \le 3$. The spacewalker says which of the two indices to drop; its strategy at the moment is to try all possibilities and rely on Pareto filtering to pick the one that yields the better result. Next, we assign a *cluster* of virtual processors to each physical processor. The cluster is always a rectangular subset of the VPs. In FIR, virtual processor v is assigned to physical processor $p \equiv v \div 2$. We denote the number of virtual processors to a physical processor by the letter γ, here equal to two.

Thus, within each tile, processor zero is responsible for the iterations $(j_1, 0)$ and $(j_1, 1)$, and processor two gets $(j_1, 2)$ and $(j_1, 3)$. See Figure 3.

We feel that this kind of orthogonal projection of the iteration space to the processor space is the right thing to do in practice, and that an oblique projection would be unnecessarily complicated.

3.2.5 Iteration schedules

We now consider the schedule. Eventually, we determine the precise cycle time for every operation that occurs. But we do this in two stages.

The schedule of operations is a shifted-linear schedule: iteration $\vec{j}$ will be started at time $\vec{\tau} \cdot \vec{j}$ where $\vec{\tau}$ 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 $\vec{\tau} \cdot \vec{j} + \rho_o$, where the scalars ρ_o will later be determined by a software pipeliner. We refer to $\vec{\tau}$ as the *iteration schedule*, and ρ as the *operation schedule*. Here, we consider the iteration schedule. Note that for a singly nested loop on one processor, $\vec{\tau}$ is just the initiation interval.

In the FIR example, we schedule iteration $\vec{j}$ at time $\vec{\tau} \cdot \vec{j} \equiv \tau_1 j_1 + \tau_2 j_2$. PICO-NPA must choose this scheduling vector $\vec{\tau}$ so as (i) to get the job done quickly, but (ii) not overload a processor by scheduling two of its iterations simultaneously. It must honor the flow dependence constraint (iii):

iteration $\vec{j}$ must come later than iteration $(j_1, j_2 - 1)$ if these are in the same tile. And (iv) we also do not want all of the uses of a read-only array element to be simultaneous, in order to avoid having to broadcast such an element to all the processors. In the present example, PICO-NPA chose $\vec{\tau} = (2, 3)$. The first component of $\vec{\tau}$ was determined by the requirements (i) and (ii). The second component was constrained (requirement (iii)) to be at least two to allow for one cycle for the addition and one for communicating a value of `y` between the two processors. The schedule $\vec{\tau} = (2, 2)$ was ruled out because of requirement (ii) – it schedules two iteration per processor on even numbered cycles and no iterations on odd cycles. The quickest schedule that honors all of the constraints is the one chosen here. It is not unique. The schedule $\tau = (-2, 3)$ works just as well. As for the read-only arrays and requirement (iv), each element of `w` will move from one use to the next, a displacement in the iteration space in the direction $(1, 0)$, with a delay of two clocks. The delay is the inner product of the schedule and the displacement: $\vec{\tau} \cdot \vec{d} = (2, 3) \cdot (1, 0)$. The elements of `x` move in the direction $(-1, 1)$ with a delay of one clock.

The schedule is shown in Figure 3. Note that except for an initial transient, one iteration is scheduled per processor on every cycle.

A schedule is *tight* if in steady state it assigns one iteration to every processor periodically with period equal to the II. We have discussed in full the general problem of finding a legal, tight, efficient schedule in another report [4]. We summarize the results here [2]. We have developed an efficient algorithm for enumerating *all* of the tight schedules for a given cluster shape [4]. 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. We illustrate it here for the case of a three-deep loop nest mapped to a two-dimensional processor mesh. Assume that iteration (j_1, j_2, j_3) is projected to virtual processor (j_1, j_2); the third coordinate is projected out. Then the set of virtual processors is clustered, with clusters of shape $\vec{C} = (C_1, C_2)$ assigned to each physical processor. Then any schedule of the form

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

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

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

In the FIR example, the cluster shape is 2, and the dropped dimension is the first, so τ_1 was constrained to be either 2 or -2. This is because with an II of one, and with $2 \cdot 8192$ iterations on each processor, any tight schedule should take about $2 \cdot 8192$ cycles. And τ_2 was required to be relatively prime to 2. The dependence constraint ruled out anything less than 2, so 3 was the smallest possible choice for τ_2 that yields a causal, tight schedule.

3.3 Loop transformation

The goal of the loop transformation phase is a nest that explicitly describes the actions of each processor at each time step. Loop transformation is accomplished in four steps: tiling, load-store elimination, change of loop indices, and recurrence-based optimization. We now describe these with the help again of FIR. The tiling step has been shown earlier.

[2] The remainder of this section may be skipped without loss of continuity.

3.3.1 Load/store elimination for uniform dependence arrays

In the tiled FIR code of Section 3.2.3, there are four array references per iteration, and two iterations are scheduled on each cycle. We clearly need to promote array elements into NPA registers in order to live with the memory bandwidth budget of two accesses per cycle. PICO-NPA does this by exploiting the uniform dependence relations described above. This promotion is expressed at this stage by using array variables that are tagged with a special name, beginning with `EVR_`, for later realization as registers in the NPA. The loads and stores will happen at iterations located at the edges of the tile. From this point, we will omit the sequential loops over tiles from our example nests.

```
/* After Load-Store Elimination */
for (j1 = 0; j1 <= 8191; j1++)
  for (j2 = 0; j2 <= 3; j2++) {
    EVR_w[j1][j2] = EVR_w[j1 - 1][j2];
    if (j2 >= 1)
        tmp_y = EVR_y[j1][j2 - 1];
    else
        tmp_y = y[j1];
    if (j2 >= 1 & j1 <= 8190)
        EVR_x[j1][j2] = EVR_x[j1 + 1][j2 - 1];
    else
        EVR_x[j1][j2] = x[j1 + 4 * j2T + j2];
    EVR_y[j1][j2] = tmp_y + EVR_w[j1][j2] * EVR_x[j1][j2];
    if (j2 == 3)
        y[j1] = EVR_y[j1][j2];
  }
```

Note the loads and stores to the memory based arrays (`x, y, w`) at the edge of the tile under guards that test for proximity to the edge. For most iterations, values are propagated from other iterations that occur earlier in the schedule. Note too that the loads of `w`, which are confined to the narrow bottom of the tile, have been distributed out to the host processor's code, to get rid of a memory port that would be little used in the hardware: the values of `EVR_w[0][j2]` need to be downloaded before the NPA starts.

We can create an EVR and perform load/store elimination only when we can precisely determine which occurrence of which left-hand side reference to an array is the source of data for every occurrence of every right-hand side reference. In particular, we can allow several LHS references to an array if we can determine, using Omega, that they cannot alias. We allocate a separate EVR for each such LHS reference. Two identical LHS references in separate branches of an IF construct are not really separate occurrences; we treat them as one LHS with a condition computation of the value to be stored. The most common case in which this is possible occurs when there are several affine LHS references having the same linear part, but different constant offsets, for example, references to `x[2*i]` and `x[2*i + 1]`. Two such references do not alias if the difference in their constants is not in the integer span of the linear part, as is the case here. Another example is the pair of references `x[i]` and `x[i + 100]` in a loop with bounds $0 \leq i < 100$. If Omega can determine, for every RHS occurrence of the array, the unique LHS occurrence that can be the source of the data, we can proceed. This technique generalizes earlier work on load elimination for read-only arrays of Weinhardt and Luk [20].

Other systolic synthesis approaches, notable the Alpha language and synthesis system [17], allow a system of affine recurrence equations (SARE) as input. The distances of flow dependences in a SARE are not necessarily constant; rather they may be affine functions of the loop indices, as for example when iteration $2i$ depends on iteration i. Techniques for conversion of affine recurrence equations to uniform recurrence equations have been developed by several researchers [19, 18].

3.3.2 Load elimination for affine, read-only arrays

An element of a read-only affine array is used in a set of iterations that lie in an affine subset of the iteration space. We pipeline these values from edges of the iteration space, following a direction that connects uses of the same array element.

Here are the details[3]. Consider the affine reference $A[F\vec{j} + \vec{f}]$, where A is a read-only array $\vec{j}$ is the loop index vector, and F is an $m \times n$ integer matrix and $\vec{f}$ is an m-vector. If $\vec{n}$ is a null vector of F (*i.e.,* $F\vec{n} = 0$) then any two iterations whose indices differ by $\vec{n}$ use the same element of A.

If F is nonsingular, there is no reuse: a load is required for every reference. Otherwise, we propagate values of A along directions given by the vectors of a carefully chosen integer basis for the null space of F.

When the null space of F is one dimensional, it has a unique (up to sign) shortest integer vector, which we use. Its sign is chosen after scheduling, so that it is directed from earlier to later iterations. Recall that we constrain the schedule so that the null space of F is not orthogonal to the scheduling vector; therefore, this appropriate choice of sign exists.

When the nullity of F exceeds one, there is a great deal of reuse of the elements of A: the set of iterations using a given element is two or more dimensional. In some cases we can achieve the smallest possible number of loads and "perfect" reuse of the data, but in others, we judged it to be too complicated to do so. There is no unique integer basis for the null space, and we have a choice to make.

We can achieve perfect reuse for the reference `A[j1]` in a loop nest of depth three. Here, $F = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$. We take the null basis consisting of the last two columns of the identity of order three. Using this basis, we generate the code:

```
if (j2 > 0)
    EVR_A[j1][j2][j3] = EVR_A[j1][j2 - 1][j3];
else
    if (j3 > 0)
        EVR_A[j1][j2][j3] = EVR_A[j1][j2][j3 - 1];
    else
        EVR_A[j1][j2][j3] = A[j1];
```

So the elements of the one-dimensional array `A` enter the tile along the set of iterations $\vec{j} = (j_1, 0, 0)$, move out along the face $\vec{j} = (j_1, 0, j_3)$, and thence into the interior. The thing that makes this perfect reuse and minimization of loads possible is that we have found a null basis consisting of vectors that are "structurally orthogonal" – they have nonzeros in different positions.

Such a nice null basis may not exist. Consider the reference `A[j1 + j2 + j3]`. Again the rank of F is one and the nullity is two. It is easy to see that F has no structurally orthogonal null basis. Suppose we propagate values into the interior along the direction $n_1 \equiv (0, 1, -1)$, which is as nice a null vector as one can find for F. Suppose the three loops all iterate ten times. Starting from an arbitrary iteration (j_1, j_2, j_3), and moving backwards in the direction $-n_1$, we will reach the boundary of the iteration space when either $j_2 = 0$ or $j_3 = 9$. We could perform loads there, giving us the simple code

[3]This section may be skipped without loss of continuity.

```
if (j2 > 0 & j3 < 9)
    EVR_A[j1][j2][j3] = EVR_A[j1][j2 - 1][j3 + 1];
else
    EVR_A[j1][j2][j3] = A[j1 + j2 + j3];
```

But this would mean reading the same value at multiple locations on these two faces. If we want perfect reuse and minimum memory traffic, we must choose a second null vector. The problem is that the appropriate second null vector depends on which face we hit. For the face $j_2 = 0$ we would want one that lies on that face, namely $n_2^{(2)} = (1, 0, -1)$. For the face $j_3 = 9$ we want $n_2^{(3)} = (-1, 1, 0)$. The code becomes complicated:

```
if (j2 > 0 & j3 < 9)
    EVR_A[j1][j2][j3] = EVR_A[j1][j2 - 1][j3 + 1];
else
    if (j2 == 0)
        if (j1 > 0 & j3 < 9)
            EVR_A[j1][j2][j3] = EVR_A[j1 - 1][j2][j3 + 1];
        else
            EVR_A[j1][j2][j3] = A[j1 + j2 + j3];
    else  /*  j3 == 9 */
        if (j1 > 0 & j3 < 9)
            EVR_A[j1][j2][j3] = EVR_A[j1 + 1][j2 - 1][j3];
        else
            EVR_A[j1][j2][j3] = A[j1 + j2 + j3];
```

Because of this difficulty, and because it arises seldom if ever in practice, we do not implement a perfect reuse strategy unless we can find a structurally orthogonal null basis for F.

3.3.3 Transforming to time and processor coordinates

Now we use the schedule and the mapping determined previously. We transform the nest, making the time step and the processor number into the loop indices. The loop body has been transformed so that it reflects what the hardware of processor p will be doing at time t. We use the notation c for the position of the active VP relative to the origin of its cluster. For the FIR example, $c \equiv v \bmod 2$.

```
for (t = 0; t <= 16391; t++)
  for (p = 0; p <= 1; p++) {
 1  j1 = <method explained below>
 2  c = <method explained below>
 3  EVR_w[t][p] = EVR_w[t - 2][p];
 4  if (0 <= j1 & j1 < 8192)
 5    {
 6      if (j2 >= 1)
 7        if (c == 0)
 8          tmp_y = EVR_y[t - 3][p - 1];
 9        else
10          tmp_y = EVR_y[t - 3][p];
12      else
13        tmp_y = y[j1];
14      if (j2 >= 1 & j1 <= 8190)
15        if (c == 0)
16          tmp_3 = EVR_x[t - 1][p - 1];
17        else
18          tmp_3 = EVR_x[t - 1][p];
19        EVR_x[t][p] = tmp_3;
20      else
21        EVR_x[t][p] = x[j1 + 4 * j2T + j2];
22      EVR_y[t][p] = tmp_y + EVR_w[t][p] * EVR_x[t][p];
23      if (j2 == 3)
24        y[j1] = EVR_y[t][p];
    }
  }
```

The bounds on the t loop are derived directly from the schedule – they are the smallest and largest values attained by $\vec{\tau} \cdot \vec{j}$ as $\vec{j}$ varies over one tile. The p loop traverses the spacewalker-specified array of physical processors. By construction, the processor loops are parallel – all dependences are carried by the outer t loop.

We are using a tight schedule that assigns one iteration of the tile to every processor at every cycle. Therefore, we can associate a point in the iteration space of the tile with every pair (t, p). Its iteration space coordinates are needed in several places, so they must be computed at line 1. (Details on how follow in Section 3.3.4 below.) Similarly, the position of the active virtual processor within the cluster assigned to a physical processor is needed and is computed at line 2 (again, details below).

Note that there are times at which the processor is idle – for example, processor one at times zero and one. The guard at line 4 tests whether processor p has a valid iteration to perform at time t.

The major change to the loop body is the reindexing of the EVR arrays. On the left-hand side they are indexed, as in the last case, by the loop indices. The occurrences on the right hand side reflect, through the constants added to t and p in the indices of the EVR arrays, the delay in time and the displacement between processors between the computation of a value and its use. For example, the use of `EVR_y[t - 3][p - 1]` at line 8 is the transformation of the use `EVR_y[j1][j2`

`- 1]` at line 3 in the previous nest. The time difference (-3) is derived from the distance $(0, -1)$ in the iteration space and the schedule $\vec{\tau} = (2, 3)$. Furthermore, the displacement by $(0, -1)$ in the iteration space is a displacement by -1 in the virtual processor index. This takes one to another physical processor if the currently active virtual processor has coordinate $c == 0$, hence the if-then-else construct in lines 7 – 10.

The abbreviation EVR stands for *expanded virtual register*. In the hardware realization of an EVR, the number of registers per processor depends only on the maximum time delay in the program. In this example, `EVR_y` requires three physical registers, `EVR_x` requires one, and `EVR_w` requires two.

We can now explain how tiling reduces register requirements. Clearly, the size of the tile has a direct impact on the number of virtual processors, and therefore on the size γ of the cluster of VPs per physical processor: the tile's shape in the dimension that is projected away doesn't matter, but its shape in the other dimensions are directly proportional to γ. The cluster size γ has a strong influence on the schedule $\vec{\tau}$. Indeed, one element of $\vec{\tau}$ is equal to γ or $-\gamma$. The others may be forced to be large if γ is large, due to dependence constraints. The time delay engendered by a dependence with distance $\vec{d}$ in the iteration space is $\vec{\tau} \cdot \vec{d}$, and so a schedule with large components is likely to produce large delays, and hence large physical realizations of the EVRs. For example, if we take the tile in FIR to have size $(8192, 16)$ (*i.e.,* we don't really do any tiling) then we get $\gamma = 8$ and the schedule $\vec{\tau} = (8, 3)$; the register count for `EVR_w` goes from two to eight, and for `EVR_x` from one to five.

Another example of a set of registers whose count scales with γ occurs in Section 3.3.4.

3.3.4 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 cycle, this code computes the coordinates (the loop indices, such as `j1` in FIR) of the iteration being started, the addresses to be used in global memory accesses, the position of the active virtual processor within the cluster (`c` in FIR), and some comparisons to determine if a datum is to be read or written to global memory or communicated from a neighboring processor (the guards at lines 6, 7, 14, 15, and 23 in the loop nest of the last section). 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 are live-in and must be downloaded. At each cycle, 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 less than or equal to the cluster size γ, so they may be obtained by downloading the first γ of their values into a ring buffer. In particular, a test will be periodic if it involves only the iteration space coordinates from dimensions other than the one projected out in the mapping. This scheme has been fully described in our report on tight scheduling [4].

We give the FIR code, after this optimization, below. The recurrence distance used here was three. Note the "decision tree" (of depth one, here) at the start of the loop body. It can be understood by examining the schedule, Figure 3. Moving ahead three cycles on a given processor means either a move in the iteration space in the direction $(0, 1)$ (done when the previous active VP has local coordinate c equal to 0) or in the direction $(3, -1)$ (when the previous active VP was from the top row, with c equal to 1).

The FIR code, after this transformation, is:

```
for (t = 0; t <= 16391; t++)
    for (p = 0; p <= 1; p++) {
        /*  The Decision Tree */
        if (EVR_c[t - 3][p] == 0) {
            del_c = 1; del_j1 = 0; del_addr_y = 0; del_addr_x = 4;
        } else {
            del_c = -1; del_j1 = 3; del_addr_y = 12; del_addr_x = 8;
        }
        /*  Updating coordinates and memory addresses */
        j1 = del_j1 + EVR_j1[t - 3][p];
        c = del_c + EVR_c[t - 3][p];
        addr_y = del_addr_y + EVR_addr_y[t - 3][p];
        addr_x = del_addr_x + EVR_addr_x[t - 3][p];

        /*  Rotating the ring buffers for periodic predicates */
        pred_0 = EVR_pred_0[t - 2][p];    pred_1 = EVR_pred_1[t - 2][p];
        pred_2 = EVR_pred_2[t - 2][p];

        EVR_w[t][p] = EVR_w[t - 2][p];
        if (j1 >= 0 & j1 < 8192)
          {
            if (pred_1)
                if (pred_0)
                    tmp_y = EVR_y[t - 3][p - 1];
                else
                    tmp_y = EVR_y[t - 3][p];
            else
                tmp_y = *addr_y;

            if (pred_1 & j1 <= 8190) {
                if (pred_0)
                    tmp_3 = EVR_x[t - 1][p - 1];
                else
                    tmp_3 = EVR_x[t - 1][p];
                EVR_x[t][p] = tmp_3; }
            else
                EVR_x[t][p] = *addr_x;

            EVR_y[t][p] = tmp_y + EVR_w[t][p] * EVR_x[t][p];

            if (pred_2) *addr_y = EVR_y[t][p];
          }
        EVR_pred_0[t][p] = pred_0;    EVR_pred_1[t][p] = pred_1;
        EVR_pred_2[t][p] = pred_2;
        EVR_addr_x[t][p] = addr_x;    EVR_addr_y[t][p] = addr_y;
        EVR_c[t][p] = c;              EVR_j1[t][p] = j1;
      }
```

Chen [3] and Rajopadhye [14] 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 no 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 predicate. As an additional optimization, we could use the scheme of Chen or Rajopadhye for nonperiodic affine predicates.

4 The back end: from parallel loop nest to hardware

Iteration-level transformations bring the loop nest into a form ready for hardware synthesis. In the next steps, the computation described by the body of the transformed loop nest is mapped and scheduled to a single nonprogrammable datapath that achieves the required II. The specified number of instances of these processors are then interconnected into a systolic array. We briefly describe the steps of this process below.

4.1 Word width

The first step towards hardware realization is to minimize the width of each integer 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. Literal constant have an inherent width. We use this information to infer width requirements for all integer data, both named variables and compiler-generated temporaries, and for 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 [10].

4.2 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 $\vec{c} = [c_f]$, 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 fully pipelined and can accept one new operation every cycle;[4] the operation latency need not be one.

The transformed loop body is characterized by a count vector $\vec{k} = [k_o]$ 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 $\vec{n} = [n_f]$. We would like to minimize the price $\vec{c} \cdot \vec{n}$ 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

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

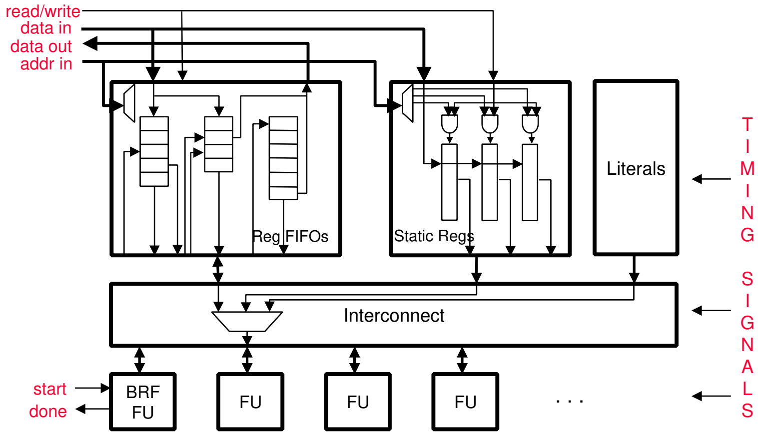

Figure 4. The processor datapath

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:

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

$$\begin{aligned} \sum_o x_{of} &\leq n_f \lambda \quad \text{for each FU type } f \\ \sum_f x_{of} r_{of} &\geq k_o \quad \text{for each operation type } o \end{aligned}$$

where λ is the required initiation interval.

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 distributing processing cycles 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 (an industry-standard ILP solver) takes about 16 seconds on a Pentium PC to solve a problem with 40 FU types and 30 operation types.

Note that there is no guarantee that we can achieve the required II with this set of FUs; we may need to allocate more at a later stage. We are, however, guaranteed that no cheaper collection of FUs can work.

4.3 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 [16]. Even though the final hardware is supposed to be nonprogrammable, at this stage we assume a fully interconnected abstract architecture which is completely programmable. This allows our re-targetable VLIW compiler, Elcor, 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, nonprogrammable architecture.

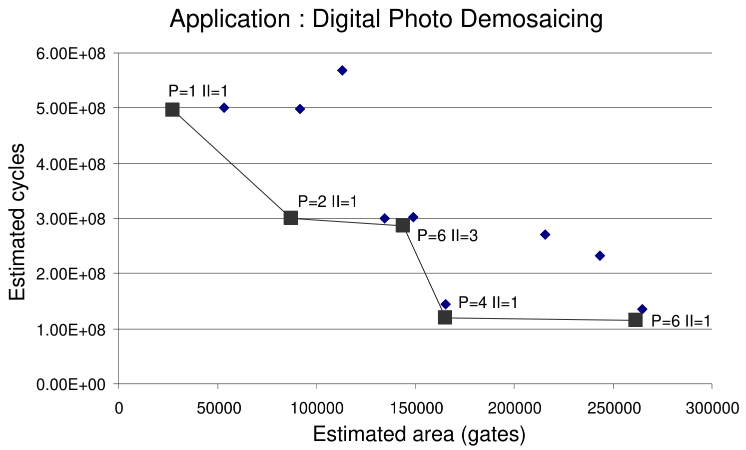

Figure 5. Example Pareto Set

The VLIW compiler performs modulo-scheduling [15] of the transformed loop body at the required II. It employs several heuristics to control hardware cost. The width heuristic reduces cost by mapping similar width operators to the same FU [10]. 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 several 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 required 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 required II.

4.4 Datapath generation

The processor datapath schema is shown in Figure 4. The various operands of the loop body are materialized using synchronous register FIFOs to hold the EVRs, registers for the static virtual registers, and hardwired literals. When the II is greater than one, several opcodes map to each of the FUs, and several sets of 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. Two EVRs can coexist in a FIFO if they are not written on the same cycle, as occurs if the operations that write to them are scheduled on the same FU at different cycles.

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.

4.5 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 [9]. 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.

4.6 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 1. The processors may share array-level local memory via dedicated local memory ports. Since memory bandwidth is constrained externally, the processors communicate with the memory through a controller that 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 Verification

The code and hardware generated by PICO-NPA is tested against input data that the user provides, automatically. These test occur at three stages: in the front end (after Phase 3), the back end, and after generation of the HDL.

In the front end, the parallel loop nest is generated as C code, compiled and linked with the original host code and any additional host code that the front end emits, and is run, and its output is compared with the original code's. This tests the functional correctness of the parallelization.

In the back end, we use the Trimaran simulator [1] to test the code. This simulation provides functional testing as well as creating test data for use in the HDL testing.

Finally, we test the generated HDL, using an HDL test bench created specifically for the artifacts that PICO-NPA generates. The tests are driven by the test data provided by the back end simulation.

6 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 from 1 to 4. Figure 5 shows the resulting cost/performance tradeoff with the Pareto points marked by their configuration. The unit II designs are particularly effective for this application.

The second experiment examines in more depth the effect of increasing the II on cost for an image sharpening application. Table 1 presents cost breakdowns for three single-processor designs, with an II of one, two, or four. (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. When the II increases from one to two, a modest cost reduction is obtained by reducing the function unit and register cost by large amounts. A side effect of increasing the 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 the 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 the II. If slower, less expensive multipliers were available in the macrocell library, these would be chosen at the higher IIs.

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

We have also taken our automatically produced designs through industry-standard ASIC and FPGA design flows for a few applications, *e.g.,* Discrete Cosine Transform, FIR filters, ATM cell delineation, and Viterbi decoding. For the IS-95 Viterbi decoder with a coding rate of one half and a constraint length of nine, PICO produced a single processor design with an II of one, whose hardware implementation required 50K gate-equivalents of area and achieved a 400 MHz clock frequency on a 0.18μ fabrication process. We were able to achieve such a high clock frequency due to the highly pipelined nature of the design. Furthermore, no significant wire routing issues were encountered during the place-and-route process; we believe this to be due to the fact that our design has mostly point to point net connections. We expect these properties to hold for most of PICO-NPA designs.

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 14% 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.

7 Related work and conclusions

The basic schema for systolic synthesis described by Quinton and Robert [13] underlies the architecture of PICO-NPA. The IRISA system [21], 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 [7] 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 [5].

There are also some practical limitations to the working version of PICO-NPA. PICO-NPA currently restricts loop bounds to be constant. It requires the user to express his computation as a perfect nest. In order to allow more that one processor to be used, the code must permit dependence analysis that discovers loop-level parallelism. To minimize the use of memory bandwidth, either local memory pragmas must be used or uniform flow dependences are required.

While earlier systems have individually incorporated some of its features, PICO-NPA goes beyond previous efforts directed at NPA compilation. It breaks new ground in its loop-level scheduling and parallelization techniques, its memory bandwidth reduction techniques, and its dedicated processor synthesis techniques. Its transformation from C code to correct, verifiable, synthesizable HDL is complete and fully automatic. It is the first such system with all of the following characteristics. PICO-NPA exploits loop-level and instruction-level parallelism; it performs both loop parallelization and efficient pipelined datapath synthesis; using advanced techniques for dependence analysis, it accepts restricted C code rather than a system of uniform recurrence equations as input; it 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; it uses advanced techniques for load-store elimination and pipelining of input data; it implements a new, practical method for systolic array scheduling; it uses a new, efficient method of code generation for parallel loops; it allows the user to extend C with the use of pragmas for data width specification and for locating arrays in local memory; it uses new techniques for data width specification, inference, and exploitation to minimize datapath cost; it uses a new resource allocation technique for low-cost, special-purpose datapath synthesis; and it uses new variations of modulo scheduling for hardware-cost-sensitive software pipelining.

PICO-NPA can be extended and improved in several directions. In our view, the most interesting and important are parallelization of an imperfect loop nest with a nonhomogeneous processor array, and the ability to take a C specification in the form of multiple loop nests, for generating a pipeline of NPAs with optimized buffers between them.

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

References

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

[2] Shail Aditya, B. Ramakrishna Rau, and Vinod Kathail. 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.

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

[4] Alain Darte, Robert Schreiber, B. Ramakrishna Rau, and Frédéric Vivien. Constructing and exploiting linear schedules with prescribed parallelism. *ACM Transactions on Design Automation for Electronic Systems*, to appear, 2001.

[5] 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.

[6] Kyle Gallivan, William Jalby, and Dennis Gannon. On the problem of optimizing data transfers for complex memory systems. In *Proceedings of the 1988 ACM International Conference on Supercomputing*, pages 238–253, 1988.

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

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

[9] Vinod Kathail, Mike Schlansker, and B. Ramakrishna Rau. HPL PlayDoh architecture specification: Version 1.0. Technical Report HPL-93-80, Hewlett Packard Laboratories, February 1994.

[10] Scott Mahlke, Rajiv Ravindran, Michael Schlansker, Robert Schreiber, and Timothy Sherwood. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, to appear, 2001.

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

[12] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[13] Patrice Quinton and Yves Robert. *Systolic Algorithms and Architectures*. Prentice Hall International (UK) Ltd., Hemel Hempstead, England, 1991.

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

[15] B. Ramakrishna Rau. Iterative modulo scheduling. *International Journal of Parallel Processing*, 24:3–64, 1996.

[16] B. Ramakrishna Rau, Vinod Kathail, and Shail Aditya. Machine-description driven compilers for EPIC and VLIW processors. *Design Automation for Embedded Systems*, 4:71–118, 1999.

[17] Tanguy Risset. The Alpha homepage. http://www.irisa.fr/cosi/ALPHA/welcome.html.

[18] V.P. 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.

[19] Vincent van Dongen and Patrice 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.

[20] 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.

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

[22] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tijang, S-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM Sigplan Notices*, 29:31–37, December 1994.