Array Control for High Performance SIMD Systems[‡]

Martin C. Herbordt
Department of Electrical and Computer Engineering
336 Photonics Center; 8 Saint Mary's Street
Boston University, Boston, MA 02215
email: herbordt@bu.edu; phone: (617) 353-9850

Jade Cravy
GDA Technologies
2071 Junction Ave.; San Jose, CA 95131

Honghai Zhang
Department of Electrical and Computer Engineering
University of Iowa
Iowa City, Iowa 52242

Note: This article has been accepted for publication in the Journal of Parallel and Distributed Computing as of September 29, 2003.

Abstract: Although arrays of SIMD PEs can be built with very high operating frequencies, problems exist in keeping the array busy. The inherent mismatch between host and array makes it difficult to maintain high array utilization: either the rate of instruction issue is very low or PE data locality is compromised, having the same effect. Our solution is based on an array control unit (ACU) design that expands macroinstructions in two stages, first by data tile and then into microinstructions. The expansion itself solves the issue problem; decoupling the expansion modalities maintains data locality. Several issues involving host/ACU interaction need to be resolved to effect this solution. We present experimental results showing that our approach delivers substantial improvement in memory hierarchy performance: a cache of only one fourth the size is sufficient to achieve the same performance as previous approaches. We also describe our implementations which demonstrate that achieving gigaherz operating frequencies with current technologies is plausible.

[*]This work was supported in part by the National Science Foundation through CAREER award #9702483, by the Texas Advanced Research Program (Advanced Technology Program) under grant #003652-952, and by a grant from the Compaq Computer Corporation.

[†]A preliminary version of this work appeared in the Proceedings of the 5th IEEE International Workshop on Computer Architecture for Machine Perception Workshop (CAMP 2000).

1 Introduction

SIMD arrays retain their niche in vision (see CAMP '97 [34] and CAMP '00 [8], e.g. [2, 5, 9, 10, 13, 15, 16, 27, 29]) and graphics (e.g. the PixelFusion graphics engine [31]). Especially promising are the prospects of SIMD arrays in systems-on-a-chip, particularly when based on SRAM cores [30], fused with CMOS sensor arrays [26], or implemented on platform FPGAs (see below). Other uses include network processors [11], signal and image processing [33], and bioinformatics [19].

One reason is that SIMD arrays can be built with large numbers of processing elements (PEs) on a single chip to deliver tremendous per-cycle processing capability. Another reason is that it is straightforward to build PEs with very high operating frequencies: except for clock and instruction issue, all of the paths can be made very short. And clock distribution and instruction issue can be optimized with the same methods that are used to distribute analogous signals in high-performance microprocessors (see, e.g. Bolotski [4] and Herbordt et al. [23]).

The basic problem in constructing SIMD array-based systems is that, while PEs can be built to be very fast and while instruction issue and clock can keep up with the PEs, *determining* which instruction to issue next can be significantly more difficult. This is a consequence of the asymmetric, multithreaded nature of SIMD program execution: a host executes the serial (main) thread, including performing the scalar operations and determining program flow, while the array of PEs executes the parallel thread as determined by the host. Since the PEs have been relieved of the control and bookkeeping responsibilities, it is often the case that ten or more serial instructions are executed for every parallel instruction. This is one of the fundamental advantages of SIMD arrays. It also leads to a serious problem: Even with the high-speeds of modern microprocessors, this scenario could easily leave a PE array idle most of the time.

One improvement that has been used previously (e.g. by Gealow [18]) is *instruction expansion*. Balance between host and array is achieved not by having the host send every PE instruction to the array, but rather, only having the host send *macroinstructions*. These macroinstructions are then expanded—by the array control unit (ACU) on the PE chip—to from several to several hundred PE instructions. The expansions themselves can be either stored in a microstore on the PE array chip, expanded on-the-fly, or a combination of the two.

This simple instruction expansion scheme, however, usually has a major negative consequence: *Much of the locality within the PE array's data stream is lost.* To explain how this occurs, we first describe how instruction expansion is possible in the first place. There are two factors, standard microcoding and tiling. The first is analogous to a microcoded uniprocessor: if a 32 bit add instruction is executed on a PE with an 8 bit ALU, then it must be expanded to at least 4 PE instructions. The second results from a mismatch between number of elements in a parallel variable (e.g. the number of pixels in an image) and the actual number of PEs in the array. This mismatch is often dealt with by mapping each element (e.g. pixel) to a *virtual* PE (or VPE). Each physical PE then emulates the appropriate number of VPEs [25]. Note that fine-grained arrays have more expansion due to the first factor and less due to the second than coarse-grained systems. In either case, expansions of a 1000-1 or more are common for current practical systems. Of this, at least 64-1 is due to tiling.

The locality problem follows from the need to emulate VPEs. Executing each macroinstruction in its entirety before continuing to the next macroinstruction (i.e., through all tiles) can multiply the size of the working set by up to the tiling factor. Since the working set is likely to be large even without VPE emulation (in bytes if not in number of images), the consequence is that PE data is unlikely to remain in the higher levels of PE memory (e.g. registers, on-chip SRAM, or cache) long enough to take advantage of the previous access. This effectively limits the latency of each PE instruction to the memory access time. Although such a bottleneck is undesirable in any system with a memory hierarchy, it is especially troublesome for an inherently pin-limited system such as a massively parallel array-on-a-chip.

In systems where the host generates and transfers all PE instructions directly to the PE array–that is, with no expansion–the loss-of-locality problem is avoided by executing PE instructions "within each tile" for as long as possible. Generally, this is until either inter-tile communication is necessary or a reduction hazard is reached; only then does execution begin of the instructions that operate on data in the next tile. This approach maximizes locality of data references, but sacrifices macroinstruction expansion on the PE chip.

What we describe in this article is an on-chip array control unit (ACU) that reconciles the apparently incompatible goals of enabling instruction issue at the PE operating frequency and maximizing PE data reference locality. The method is to preload entire "basic blocks" of macrocode onto the PE chip and then letting the array control unit (ACU) handle the expansion. The ACU sends PE instructions of the entire basic block for a single tile before doing the same for each succeeding tile. Locality is preserved because the execution thread remains within each tile for a significant time before that tile is "rolled out" and the next tile "rolled in."

In order to make on-chip basic block expansion work, various issues need to be addressed; their solutions–together with the experimental results showing the benefit of this approach–provide a primary contribution of this work. One issue is dealing with the dynamic address computation required by on-the-fly expansion. Another is dealing with dynamic scalar variables which are needed by some PE instructions. The values of these scalar variables are often not known until they are computed by the host *after* the basic block has been sent to the ACU and just before the scalar is needed by the array.

The overall significance of this work is that it makes viable larger SIMD arrays that can work on larger problems than would otherwise be possible. It allows, without slowing down the PE array:

- applications with significant working sets to take advantage of PE memory hierarchy,
- the host to be on a separate chip from the PE array and ACU, and
- the existence of multiple PE/ACU chips in a SIMD array system.

One problem not addressed here is, for multi-PE-chip systems, dealing with communication latency across chip boundaries. We will, however, briefly address this issue in the conclusion. In the following sections we describe, in turn: SIMD array architecture including memory hierarchy, VPE emulation and instruction expansion methods, implementation issues related to these methods, our own hardware implementations, experimental results comparing data locality of our approach with previous designs, and conclusions.

2 Architecture Review

Computer systems based on SIMD arrays are asymmetric, having a single host/controller and an array consisting of from a few hundred to a few hundred thousand PEs. The PEs execute synchronously code specified by the host. One consequence is that PEs do not have individual micro-sequencers or other control circuitry; rather their "CPUs" consist entirely of the datapath. Within this constraint, however, there are few limitations. The complexity of PE datapaths has ranged from the MGAP which did not have even a complete one-bit ALU [32] to the MasPar MP2 which contained a 32-bit datapath and extensive floating point support [3].

Another consequence of SIMD control is on PE memory configurations: Each array instruction generally operates on the same registers and memory locations for each PE. It is therefore possible to model the memory hierarchy, including cache, as that of a serial processor.[1] For more on modeling PE memory hierarchies see [24, 1].

Recently, technology has favored PE arrays based on memory chips. An especially promising approach has

[1] An exception is for PE designs with a local indexing mode where registers can point to memory locations. There are usually restrictions on the indirect mode, however, so the serial model is often adequate for this case as well.

been taken by NEC with the IMAP [17]. One version is based on an SRAM chip where half of the memory has been removed and the freed-up space used for PEs. A later generation IMAP adds control circuitry for off-chip DRAM access [16]. The Semiconductor Industry Association Roadmap, together with a back-of the envelop calculation, yields a projection of possible per-chip capabilities of successive PE chips (see Table 1). Naturally there are many design alternatives for the later generation(s) with granularity being determined by application, available pins, etc. For more on these design alternatives and their evaluation see [23].

Chip	1999	2001	2003
Production DRAM	256Mb	(512Mb)	1Gb
Production SRAM	16Mb	(32Mb)	64Mb
SRAM-based IMAP-like processor	256 16 bit PEs 8Mb SRAM avail. 4KB per PE		256 32 bit PEs w/ FP 32Mb SRAM avail. 16KB per PE

Table 1: Shown is one possible development path for SIMD PE chips.

The system model we have in mind is shown in Figure 1. The host is a microprocessor (in our work, a standard PC) and the PE chips and memory are on a coprocessor card (e.g. on a PCI bus [12]). There are some number of PE chips on the card, each with a copy of the Array Control Unit (ACU). PEs have their own memories on chip, which can be either software or hardware controlled cache. There is a great deal of flexibility in organizing off-chip memory; see [17] for one example. In the conclusion we briefly discuss the implications of systems where the host is on-chip as well.

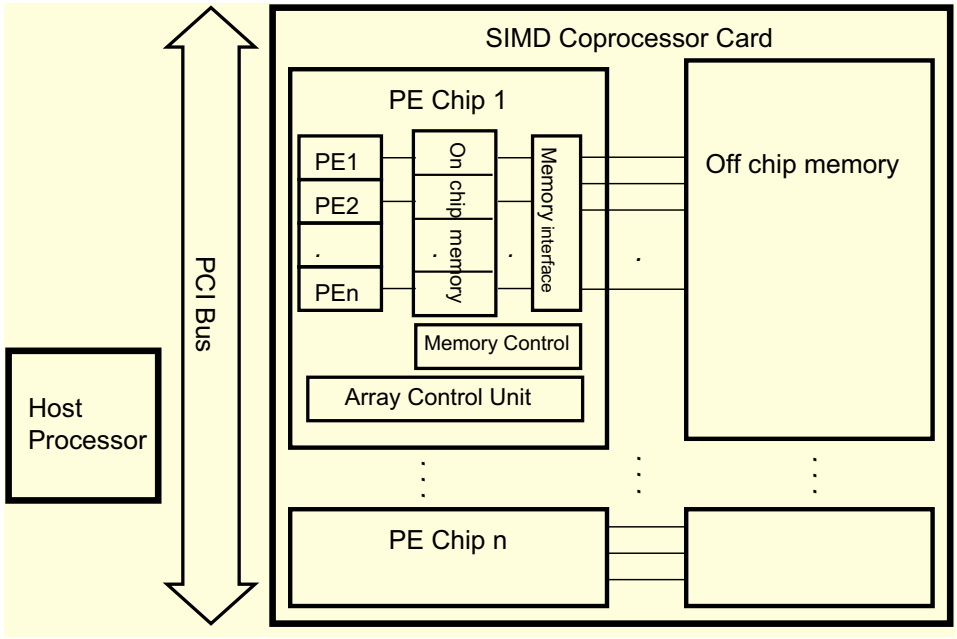

Figure 1: Shown is the implementation model with SIMD array on a coprocessor card and ACUs on the PE chips.

3 Instruction Expansion Definitions and Orderings

In this section we detail the types of expansion, their possible execution orderings, and qualitatively evaluate those orderings. We begin by describing how VPE emulation is carried out. VPE variables are mapped *VPR* elements per physical PE, where VPR refers to the VPE/PE ratio. We refer to a single physical slice of a VPE variable across the array of physical PEs as a *tile*. In Figure 2, there are 16 tiles. The reverse mapping is also plausible; in this example the result would be each quadrant of the image being mapped to one PE. There can be large benefits for each mapping, depending on network support (see [20, 28]), but this decision has little impact with respect to either instruction issue or reference locality, the two primary topics of this article.

0,0	1,0	0,1	1,1	0,2	1,2	0,3	1,3
2,0	3,0	2,1	3,1	2,2	3,2	2,3	3,3
0,4	1,4	0,5	1,5	0,6	1,6	0,7	1,7
2,4	3,4	2,5	3,5	2,6	3,6	2,7	3,7
0,8	1,8	0,9	1,9	0,10	1,10	0,11	1,11
2,8	3,8	2,9	3,9	2,10	3,10	2,11	3,11
0,12	1,12	0,13	1,13	0,14	1,14	0,15	1,15
2,12	3,12	2,13	3,13	2,14	3,14	2,15	3,15

Figure 2: Shown is an 8x8 image with 4 PEs and 16 tiles. Each pixel labeled PE,tile.

Of more concern is how the VPE variables are laid out in PE memory. There are again two possibilities as shown in Figure 3. In (a), parallel variables are allocated contiguously, while in (b) VPR variable elements (tiles) are allocated contiguously. Which one yields better spatial locality depends on expansion order (described next).

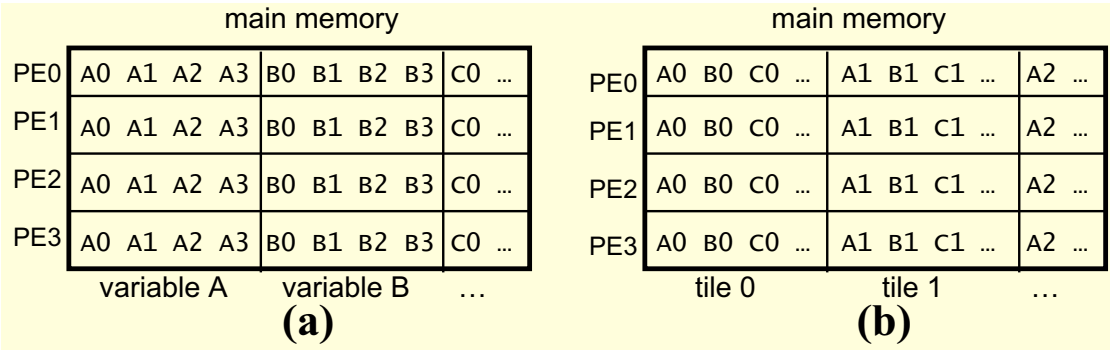

Figure 3: Shown are two possible variable layouts in memory, (a) variable first and (b) tile first, for four PEs, a VPR of 4, and some number of parallel variables.

For each VPE instruction where there is no interaction among elements *within* a parallel variable–such as those for logic, arithmetic, and activity control–VPR physical instructions must be executed, one for each tile. Emulating interacting VPE instructions such as feedback and communication is more complex and can sometimes add significant overhead [21].

The best data reference locality is achieved when as many instructions as possible are executed for each tile before continuing to the next; this is because a change of tile is effectively a context switch. However, tiles *must* be swapped whenever feedback or communication instructions are encountered. This is because feedback instructions can cause control hazards in the host and because communication instructions almost always involve interaction of elements across tiles. Compilers (e.g. ICL [35, 7, 22]) can schedule instructions to minimize tile swapping. We illustrate these ideas with a code example, beginning in a high-level data parallel language:

```
// A, B, C, D, and E are 256x256 images of 32-bit
// integer pixels stored in the PE array.
// a is a 32 bit scalar stored in the host.

IntPlane(256,256) A,B,C,D,E;
int a;

B = 10;               // 10 is a scalar known at compile time
a = 8 + 5;            // host-only operation
D = a + B;            // a is a run-time scalar
A = 7 + C;            //
if (E.ANY()) A = 1;   // ANY is a global-OR
else B = 1;           //   feedback operation
```

The following is a compiler translation of the source into tuples; `temp` is a scalar variable:

```
1. (=,B,10)
2. (+,a,8,5)
3. (+,D,a,B)
4. (+,A,7,C)
5. (ANY,temp,E)
6. (IF,temp,Tuple7,Tuple9)
7. (=,A,1)
8. (J,Tuple10)
9. (=,B,1)
10.
```

We will return to the tuple sequence later; for now we show the expansions for a single tuple: (+,A,7,C). We begin by assuming machine independence, i.e., we use the VPE array as a target and assume a VPE datapath similar to that of a standard 32-bit RISC machine. Let the 64K elements of A be assigned one per VPE and, within each VPE, to memory location 520. Let C be similarly assigned to 528. Compilation into an ordinary RISC machine assembly language yields the following VPE code. Note that these instructions belong to the array thread and so are broadcast to all VPEs. We call these VPE array assembly language instructions *VPE macroinstructions*.

```
4.1  R0 <-- 528
4.2  R1 <-- R0 + 7
4.3  520 <-- R1
```

We now begin the machine-dependent phase of compilation. Let us assume a VPR of 2 and that each VPE the PE is emulating (i.e., the tile) has been allocated 1000 bytes of memory for each physical PE. Variables A and

C now occupy 2 words each in every PE's memory, one for each tile (i.e., 520/1520 and 528/1528). The following code shows one expansion; others are possible, but require more registers. We refer to this phase as *VPE expansion* and the instructions below as *PE macroinstructions*.

```
4.1.1  R0 <-- 528
4.2.1  R1 <-- R0 + 7
4.3.1  520 <-- R1
4.1.2  R0 <-- 1528
4.2.2  R1 <-- R0 + 7
4.3.2  1520 <-- R1
```

Now assume that the PEs have a 16-bit rather than 32-bit datapaths. Then the two registers must each have two accessible halves, e.g. R0 has R0_0 and R0_1, and the following code is generated. We have now finished the compilation and that these are the actual *PE instructions*. We refer to this final phase as *ALU expansion*.

```
4.1.1.1  R0_0 <-- 528              # load half-words of PE variable
4.1.1.2  R0_1 <-- 530              #   C into PE register R0
4.2.1.1  R1_0 <-- R0_0 + 7         # add low half of 32 bit immediate
4.2.1.2  R1_1 <-- R0_1 + 0 w/ carry # add high half of 32 bit immediate with carry
4.3.1.1  520 <-- R1_0              # store half-words of register
4.3.1.2  522 <-- R1_1              #   R1 into PE variable A
4.1.2.1  R0_0 <-- 1528             # load half-words of PE variable
4.1.2.2  R0_1 <-- 1530             #   C into PE register R0
4.2.2.1  R1_0 <-- R0_0 + 7         # add low half of 32 bit immediate
4.2.2.2  R1_1 <-- R0_1 + 0 w/ carry # add high half of 32 bit immediate with carry
4.3.2.1  1520 <-- R1_0             # store half-words of register
4.3.2.2  1522 <-- R1_1             #   R1 into PE variable A
```

The final code is labeled with 4 indices, corresponding to the following transformations: (1) high-level language to tuple, (2) tuple to VPE macroinstruction, (3) VPE macroinstruction to PE macroinstruction, and (4) PE macroinstruction to PE instruction. The execution order, however, follows a different nesting; in particular (2) and (3) have been reversed. Other execution orderings are also possible: this decision process is identical to that of selecting the appropriate nesting order for, say, a matrix multiplication code, and has similarly critical importance both on data reference locality and on determining the appropriate hardware support. Also, it is possible to implement "blocking" analogous to that used in compiler optimization, e.g., when accounting for cache size or multiprocessing.

We now discuss qualitatively the various possible execution orderings. Details and experimental results are presented in the next sections. There are 24 possible index orderings, each corresponding to a permutation. Most, however, make little sense and are quickly eliminated.

1. VPE macro instructions (2) can be optimized across tuples (1) but since that is a standard compiler function we do not discuss it further here. However, a complete interchange of (1) and (2) makes little sense since macroinstruction index does not have semantic content outside the tuple. Also, this interchange generally results in many additional registers being required and for even modest VPR leads to code that will not execute. We therefore assume that (1) always precedes (2).

2. ALU expansion (4) should always come last; i.e. PE macroinstructions should not be interleaved. The following example, two versions of tuple 4, tile 2, shows why. In the first version, the three macroinstructions are executed consecutively; in the second they are interleaved. In both versions, we need to save the carry from 4.2.2.1 for use in 4.2.2.2. In the second, however, 4.2.2.1 and 4.2.2.2 are no longer contiguous and the carry must be saved across

instructions. Although this is not a problem here, in general, interleaving PE macroinstructions would require saving a set of status flags for every PE macroinstruction in progress.

```
# version 1:  PE macroinstructions 1, 2, 3 are consecutive
4.1.2.1  R0_0 <-- 1528              # load half-words of PE variable
4.1.2.2  R0_1 <-- 1530              #   C into PE register R0
4.2.2.1  R1_0 <-- R0_0 + 7          # add low half of 32 bit immediate
4.2.2.2  R1_1 <-- R0_1 + 0 w/ carry # add high half of 32 bit immediate with carry
4.3.2.1  1520 <-- R1_0              # store half-words of register
4.3.2.2  1522 <-- R1_1              #   R1 into PE variable A

# version 2:  PE macroinstructions 1, 2, 3 are interleaved
4.1.2.1  R0_0 <-- 1528              # load half-words of PE variable
4.2.2.1  R1_0 <-- R0_0 + 7          # add low half of 32 bit immediate
4.3.2.1  1520 <-- R1_0              # store half-words of register
4.1.2.2  R0_1 <-- 1530              #   C into PE register R0
4.2.2.2  R1_1 <-- R0_1 + 0 w/ carry # add high half of 32 bit immediate with carry
4.3.2.2  1522 <-- R1_1              #   R1 into PE variable A
```

This leave three possible permutations: 1234, 1324, and 3124.

3. That 1234 is also not viable is shown below. Obviously this code is incorrect and different registers would be used by the 4.*.2.* instructions. The problem is that a VPR sets of registers would be required.

```
4.1.1.1  R0_0 <-- 528
4.1.1.2  R0_1 <-- 530
4.1.2.1  R0_0 <-- 1528
4.1.2.2  R0_1 <-- 1530
4.2.1.1  R1_0 <-- R0_0 + 7
4.2.1.2  R1_1 <-- R0_1 + 0 w/ carry
4.2.2.1  R1_0 <-- R0_0 + 7
4.2.2.2  R1_1 <-- R0_1 + 0 w/ carry
4.3.1.1  520 <-- R1_0
4.3.1.2  522 <-- R1_1
4.3.2.1  1520 <-- R1_0
4.3.2.2  1522 <-- R1_1
```

The two final permutations are both interesting and are shown for two tuples:

4. In 1324, the priority is on executing VPE macroinstructions consecutively. The resulting macrocode for tuples 3 and 4 is shown.

```
# VPE Expansion First
3.1.1   R0  <-- 524      # get B from PE memory
3.2.1   R0  <-- R0 + a   # add scalar a to PE reg
3.3.1   532 <-- R0       # write back D
3.1.2   R0  <-- 1524     #
3.2.2   R0  <-- R0 + a   #   Tile 2
3.3.2   1532 <-- R0      #
4.1.1   R0  <-- 528      # get C from PE memory
4.2.1   R0  <-- R0 + 7   # add scalar 7 to C
4.3.1   520 <-- R0       # write back A
4.1.2   R0  <-- 1528     #
4.2.2   R0  <-- R0 + 7   #   Tile 2
4.3.2   1520 <-- R0      #
```

5. In 3124 the priority is on executing as many macroinstructions for each tile as possible before executing the same macroinstructions for the next tile. This requires that VPE macroinstructions be interleaved. This is generally possible; the exception is when there are dependencies *within* the VPE macroinstruction, which happens exactly when there is interPE communication or when feedback is collected (usually for a flow control decision in the host). The resulting code for two tuples is shown.

```
# Execute within VPEs (tiles) for as long as possible
3.1.1   R0  <-- 524      # get B from PE memory
3.2.1   R0  <-- R0 + a   # add scalar a to PE reg
3.3.1   532 <-- R0       # write back D
4.1.1   R0  <-- 528      # get C from PE memory
4.2.1   R0  <-- R0 + 7   # add scalar 7 to C
4.3.1   520 <-- R0       # write back A
3.1.2   R0  <-- 1524     #
3.2.2   R0  <-- R0 + a   #
3.3.2   1532 <-- R0      #    Tile 2
4.1.2   R0  <-- 1528     #
4.2.2   R0  <-- R0 + 7   #
4.3.2   1520 <-- R0      #
```

4 Implementation Issues

We are left with two viable expansion orderings: *VPE First* and *Tile First*. We discuss these now with respect to (i) hardware support for instruction issue and expansion and (ii) reduction hazards. The latter are defined as control hazards where program flow is dependent on feedback from the array, usually through a reduction operation such as ANY or COUNT.

VPE Expansion First

The host sends the VPE macroinstructions to the array as they occur in the main thread. The ACU does both expansions. See Figure 4a. Instructions are generated from each VPE macroinstruction through incremental changes to the instruction fields as shown in the previous section. The advantages of VPE First are large expansion factor and simplicity: the ACU can easily generate the PE instructions with no loss in cycle time and with only a few extra pipeline stages. The disadvantage is that poor data locality is likely. Since the array executes all instructions as they occur in the host code, handling reduction hazards requires only waiting for array-host-array turn-around.

Tile Expansion First

The host handles the tiling expansion and sends only the PE macroinstructions to the array. The ACU does only the ALU expansion, (see Figure 4b). While data locality is much improved, instruction issue is a now problem. Historically, since ALU expansions were typically 16 to 32, this was just adequate to keep the array busy. Later, large issue buffers were used to deal with issue variance. These large buffers needed to be flushed on reduction hazards. Now-a-days, however, an ALU expansion of between 1 and 4 is more likely; not nearly enough to hide the host's instruction issue overhead.

Solution – The Basic Block method

The problem is therefore that neither ordering provides both viable instruction issue and reasonable data locality. Our solution is to use the Tile First expansion but move much of the control to the ACU. We begin with some definitions. We use the compiler term *basic block* to refer to the sequences of PE instructions that can be executed within a tile before execution on a new tile must be initiated. We use the term *ACU instruction* to refer to directives by the host to the ACU, but not to the PEs.

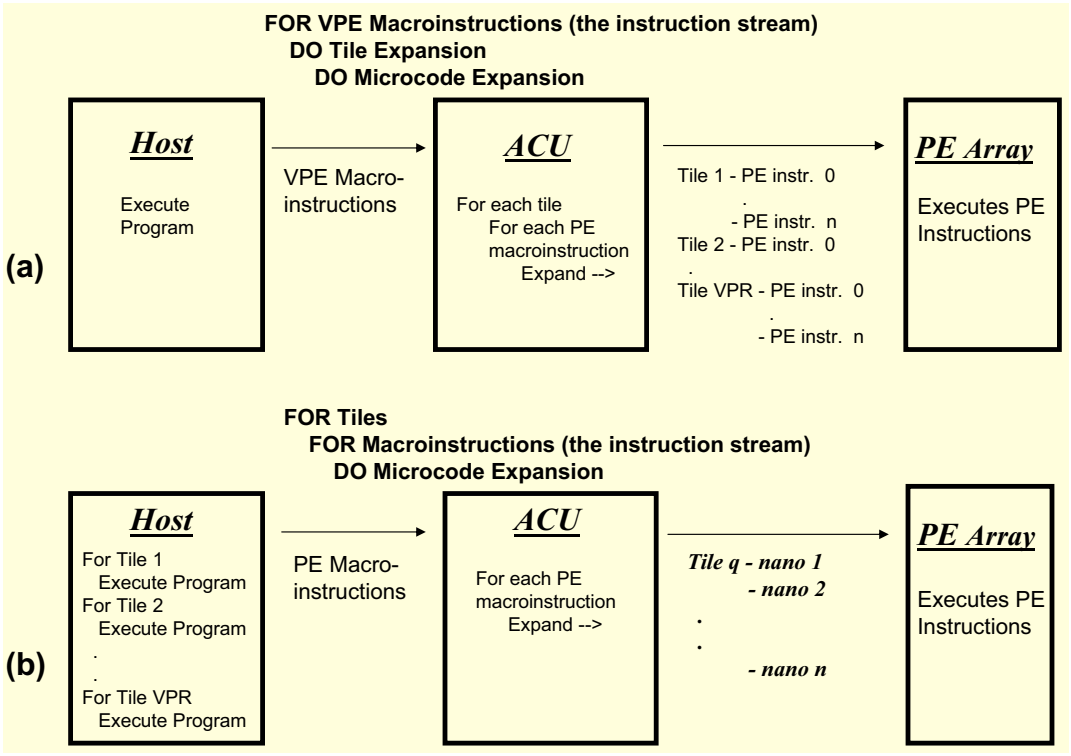

Figure 4: (a) Both expansions are done by ACU; (b) Only ALU expansion is done by ACU.

From our original example, the first five tuples form a basic block. It is shown together with the corresponding VPE macroinstructions (keeping our previous memory allocation):

```
  TUPLES          MACROINSTRUCTIONS
1. (=,B,10)       R0  <-- #10
                  524 <-- R0
2. (+,a,8,5)
3. (+,D,a,B)      R1  <-- R0 + scalar(a)
                  532 <-- R1
4. (+,A,7,C)      R2  <-- 528
                  R3  <-- R2 + 7
                  520 <-- R3
5. (ANY,temp,E)   ANY(536)
                  DONE
```

There are several things to note. One is that Tuple 3 has no corresponding macro instruction since it is executed entirely within the host. A second is that, although the scalars 10 and 7 are known before run time, the value of scalar `a` is not. Finally, the ACU needs to know that the basic block is finished so it can loop back or look for more work. See Figure 5 for a sketch.

The host begins program execution by downloading a fraction of the program's basic blocks into the ACU's macroinstruction memory. The host then executes its program. However, rather than broadcasting each PE instruction—or even each macroinstruction—as needed, it only needs to broadcast the appropriate directives. These include:

- Configuration information, such as VPE memory allocation
- The address of the next basic block

- The run-time-computed scalars and addresses for insertion into the instruction stream
- Actions to be taken associated with the basic block, such as "execute 10 times," "look for next basic block," or "examine feedback result and execute one of the two following basic blocks depending on result."

The host thus executes the following code:

```
1. ACU(CONF,2,1000);      // tell ACU there are 2 tiles of 1000 bytes
2. ACU(BB,134,1);         // tell ACU to execute BB at 134 once
3. a = 8 + 5;             // compute run time scalar
4. ACU(IMM,a);            // compute run time scalar
5. temp = ACU(FEEDBACK);  // get feedback value from ACU to temp
// do test and tell ACU which BB to do next
6. if (temp == TRUE) ACU(BB,293,1);
7. else ACU(BB,572,1);
```

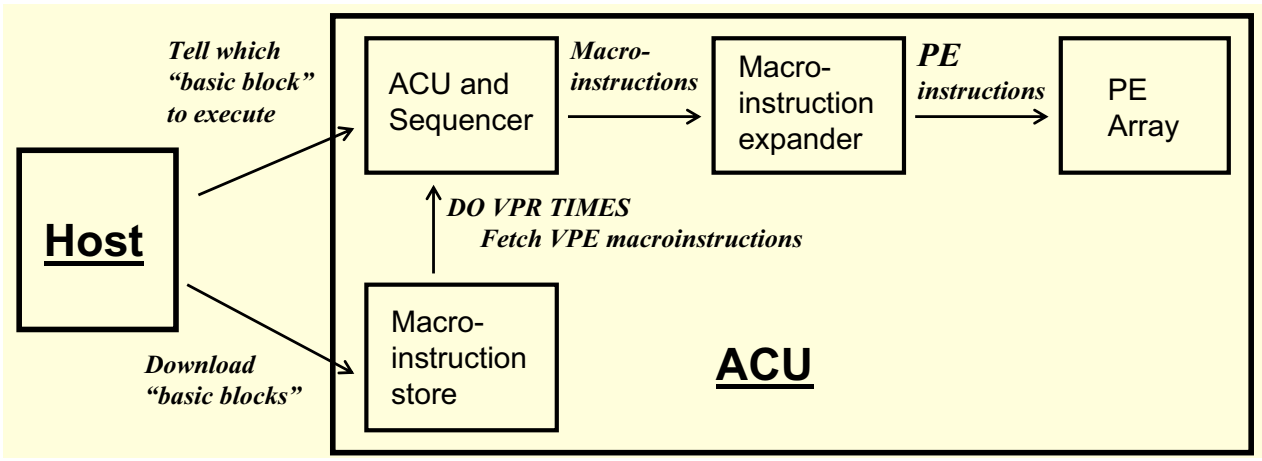

Figure 5: Basic block expansion by the ACU.

ACU() is the generic function used by the host to talk with the ACU. For efficiency, ACU() is in-lined and done with memory-mapped user-mode I/O [12]. While the host is executing the code just shown, the ACU is initially idle and waiting for instructions to come through its INFIFO. The first ACU directive, `ACU(CONF,2,1000)`, initializes the ACU's internal registers. The second begins macroinstruction fetch, expansion, and PE instruction issue. The ACU recognizes when a macroinstruction needs a run-time scalar or a dynamically computed address and reads it from its INFIFO. The ANY macroinstruction requires that the ACU keep track of the current feedback value; when the entire basic block has been executed VPR times, the feedback value is put into its OUTFIFO. Finally, the ACU looks to the INFIFO for the next basic block. The ACU can also deal with reduction hazards directly; this is described below.

5 ACU Implementation

5.1 Functions and Components

A schematic of the hardware design is shown in Figure 6. We are exploring several different implementations (see Section 5.2), but they all have the same basic characteristics. At the highest level, the system consists of a host PC which communicates via a PCI interface with the board containing the array. On the array board are some

number of ACU/PE chips, additional DRAM for PE data, and a video interface. PEs are as described in [23]: they have some amount of on-chip SRAM and an interface to off-chip DRAM. The PE granularity and memory sizes are technology and application dependent.

The keys to implementing the ACU are that the operating frequency be comparable to that of the PEs and that a PE instruction be issued every cycle. Minimizing the length of the instruction issue pipeline is also a goal, but much less important than the first two. We now enumerate some functions required for ACU operation and the hardware units that carry them out.

1. *Host sends basic blocks of macroinstructions to ACU.* Done by PCI interface and SRAM arbiter.
2. *Host sends and receives PE data.* Done by PCI interface and DRAM arbiter.
3. *Host sends ACU instructions and dynamic scalars.* Done by PCI interface and INFIFO.
4. *Host receives feedback data.* Done by OUTFIFO and PCI interface.
5. *Sequence PE macro instructions.* Done by macroinstruction fetch unit.
6. *Expand macro instructions into PE instructions.* Done in two stages: the macroinstruction fetch unit does tile expansion through iterations of basic blocks. The macro to PE code expansion is done by the macroinstruction expander.
7. *ACU instruction execution.* The macroinstruction fetch unit is the "controller controller."
8. *Dynamic scalar integration with PE instructions.* Done by the macroinstruction fetch unit.
9. *PE instruction issue.* Done by the PE instruction FIFO and the execution controller.

The core of the ACU is the Macroinstruction Fetch unit. It executes two ACU instructions: *configure* and *execute basic block*. Configure tells how many times to execute each basic block and the size of the PE memory per tile. Execute gives the address of the next basic block in the Macroinstruction SRAM. The Macroinstruction Fetch unit also handles dynamic scalars by recognizing which macroinstructions need them, fetching the values from the INFIFO, and integrating the values into the macro instructions.

The hardware details are as follows. There is a program counter for macro instruction sequencing and a tile counter for determining when execution of the basic block is complete. We assume that the size of data allocated to each PE tile is a power of two. This allows address calculation of the PE data to proceed without addition as the tile number can be merged with the macro instruction data address to generate the correct data address for each tile. The Macroinstruction Fetch unit has a three stage pipeline, including writing to the Macroinstruction FIFO.

The Macroinstruction Expansion Unit also has a three stage pipeline including writing to the PE Instruction FIFO. Its functions include sequencing the expansion and merging the register file addresses.

Two more stages are needed by the execution controller and PE decoders yielding a maximum of eight stages between macro nstruction fetch and PE instruction issue to the PE array. Depending on instruction distribution timing characteristics of the PE array, additional stages may be required there as well. Finally, the PEs themselves have three stage pipelines.

5.2 Implementation Status

We have a large number of working PE designs which can be interchanged depending on the granularity required by the application [23]. There are also a number of computer vision applications written in the data parallel language ICL [22]. Host interface software has been developed [12]. Three implementations were examined, based, respectively, on (i) a Nallatech PCI board and plug-in module which together contain a PCI interface, 3 Xilinx FPGAs (2 Virtex 1000s and 1 Virtex 300, both with -4 speed grade), 4MB SRAM, and 4MB SDRAM, (ii) a Chip

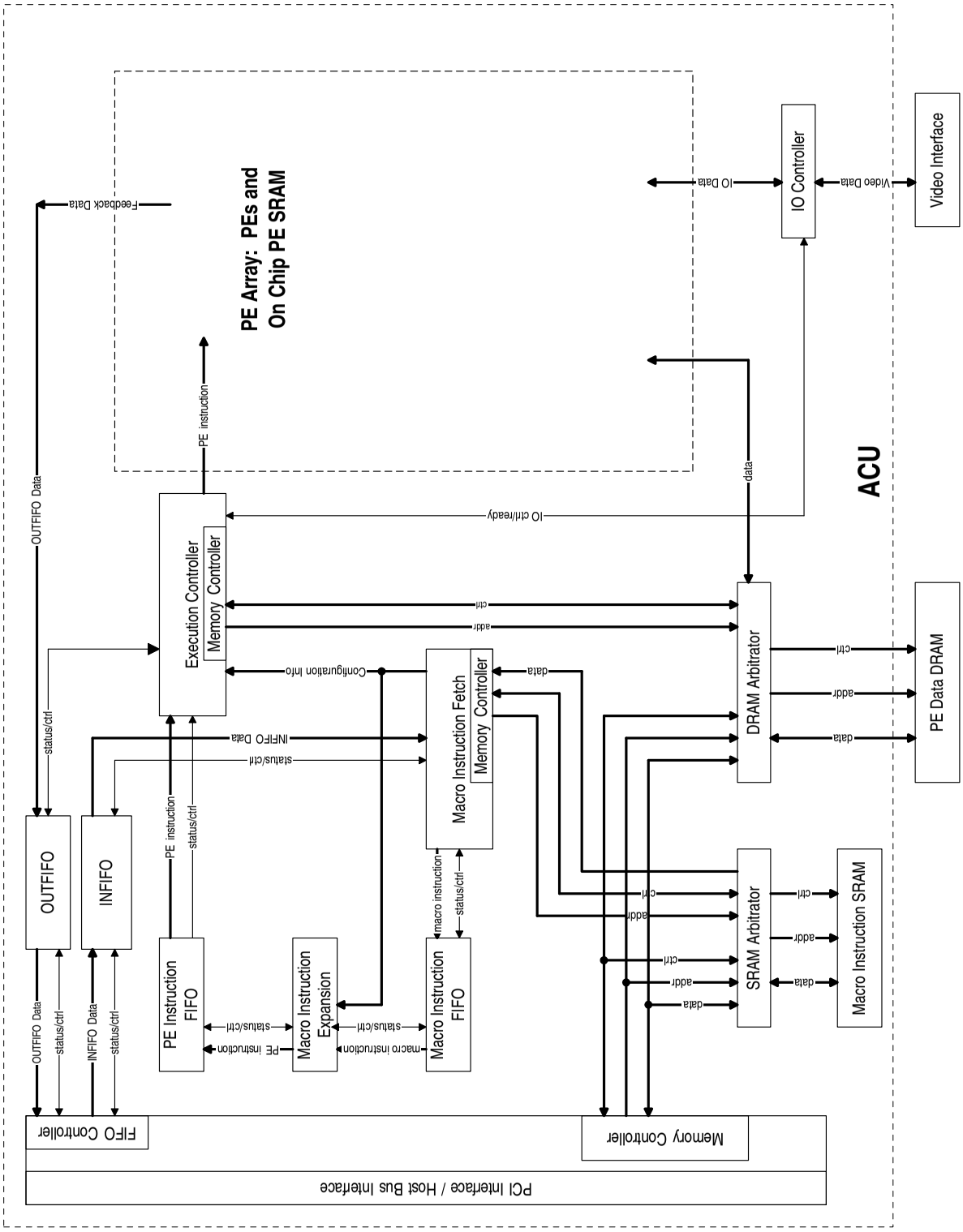

Figure 6: Shown is the block diagram for the ACU together with interfaces to host, PE array, and I/O.

Express .35 micron gate array, and (iii) a .25 micron standard cell process from LSI Logic. The Nallatech-based implementation uses a commercial platform while the Chip Express and LSI Logic implementations are virtual prototypes (synthesis only). PEs are 8 bit with multiplier and 32 bytes of register file. In the Nallatech/Xilinx system, we use the Virtex 300 and the SRAM for the ACU and the 2 Virtex 1000s and the DRAM for the PE array. For the Chip Express system, we assume the CX3551 master slice. Timing and area results are in Table 2. Operating frequencies of roughly 50 MHz, 500 MHz, and 1 GHz are indicated. Note that these technologies represent the mainstream for the year 2000.

In interpreting these numbers, the following should be considered. First some arguments indicating that the numbers are too aggressive:

- For the virtual prototypes, synthesis output does not provide accurate timings for global signals such as clock and instruction distribution.
- Place-and-route and fabrication invariably lead to reduced frequencies.

Now some arguments indicating that these numbers are too conservative:

- Almost all of the signal paths are very short.
- Distribution of clock and instructions are well understood by experienced practitioners and the issues here are not significantly different than for microprocessors.
- Several years of advances in process technology indicate a more than doubling of operating frequency.
- The designs were not subject to industry-level optimization.

	Nallatech/Xilinx 2 Virtex1K -4 FPGAs	Chip Express .35u Gate Array	LSILogic G11 .25u Standard Cell
ACU Timing	19.2ns	2.14ns	.79ns
PE Timing	21.5ns	1.95ns	.85ns
# of PEs	128 in system	64 per chip	256 per chip
per PE SRAM	128 bytes	512 bytes	50% of chip
per PE DRAM	16KB	TBD	TBD

Table 2: Characteristics of three implementations. Per PE SRAM is on chip; per PE DRAM is off chip.

6 Impact on Memory Performance

6.1 SIMD Image Processing Program Characteristics

A more realistic view of SIMD array code

For memory performance, the two critical application characteristics are the image size, from which the VPR immediately follows, and the distribution of reduction hazards and communications from which the basic block size follows.

Reduction hazards tend to be rare: where reduction hazards do occur, they are generally termination conditions of relaxation-based algorithms. Although this is an important paradigm, even here, reduction hazards rarely occur more often than once per 10's of thousands of PE instructions.

Communication, on the other hand, tends to be either frequent (occurring every few VPE macroinstructions) or rare (occurring less than every several hundred VPE macroinstructions). An application with the former characteristic is correlation-based stereo matching. An example of the latter is a focus-of-expansion computation. Some applications, e.g. a region-merging segmentation algorithm, run in phases between the two.

Since communication instructions generally include transfers *among* all tiles in a parallel variable, all preceding VPE instructions must be completed before they begin. For the same reason, communications must complete before the following instruction begins. We can therefore view each communication instruction as its own basic block.

Implications on working set size

For large basic blocks, the working set size per physical PE approaches that of a single VPE. For small basic blocks, the working set size includes the working sets of all VPEs being emulated by the PE. VPE working sets being equal, instruction sequences with small basic blocks can result in working sets up to VPR times greater than sequences with large basic blocks. Also, the bigger the VPR, the bigger the basic blocks needed for the effect of the VPR on the working set size to diminish.

Where this has the potential to be a serious problem is in applications with frequent communication, especially if the images are large (resulting in large VPR). Fortunately, it appears to be almost universally the case that applications with frequent communications operate on very few images at a time. The paradigmatic high-communication tasks are convolutions and correlations for matching: both operate on only a few images at a time. The design of memory hierarchies for high-communication tasks reduces to satisfying a simple condition: provide enough on-chip memory to hold the images being correlated. Current technology makes satisfying this condition straightforward for all but the very largest images. In this last case, however, the best solution may be to process parts of the image separately.

Implications for instruction issue

The worst case for instruction issue is where the basic block size is one. Here, however, the Basic Block Method simply reduces to VPE First. As described above, VPE First does fine with instruction issue (its problem being reference locality).

Dealing with reductions in the ACU

As with most programs, SIMD array codes are built on loops. A substantial difference with corresponding high-level language codes is that each VPE macroinstruction generally replaces the two innermost loops (rows and columns). The remaining loops in the dataparallel code therefore have the characteristics of outer loops in sequential computations: they are executed relatively few times.

It is usual that basic blocks get used several times in close succession. However, it is rare that a basic block gets used iteratively: most often, two or more basic blocks are executed in a loop with communication being interspersed. As a consequence, the ACU support for iterative execution of basic blocks is rarely used. It would certainly not be very difficult to augment the ACU to execute a basic block stream. However, there does not seem to be a reason to do so at this time by the same reasoning as immediately above.

6.2 Results and Discussion

Hardware Model

- Number of boards – We assume a single board SIMD array; more are possible, but exacerbate communication difficulties.
- On-chip memory – We assume the upper bound of on-chip memory size to be half that of an SRAM chip (with the other half of the chip allocated to PE logic). In current technology this comes to 16Mb per PE chip, growing to four times that in the next two to three years. The low-end (e.g. in an FPGA-based system) could be substantially smaller.
- Off-chip memory – With current memory densities and our applications, having sufficient off-chip memory to hold all application data should not be an issue.

- Number of PE chips – Between 1 and 8 are reasonable for a standard sized PCI board.
- Number of PEs per chip – From our own results and those from NEC, 256 8-bit PEs per chip are viable using three-year-old technology. In the next two to three years, putting 512 PEs with 32-bit datapaths and floating point support on a chip should be possible.
- Size of cache per PE – From the previous, possible cache sizes per PE range from 128 bytes with a current FPGA-based design to 16KB for a projected standard-cell design.
- Total cache size – This is an important parameter for communication intensive applications. The range for standard cell designs is 2MB for a current single chip design to 64MB for a future eight chip system. This is enough to hold from 8 to 256 medium resolution images.
- Cache parameters – A previous study has shown the benefits of high associativity and small block sizes for PE caches [24]. We therefore assume an associativity of eight and a block size of four.
- Register File Size – Because many of the parameters normally stored in registers have to do with control which a SIMD PE does not need to deal with, SIMD PEs can have smaller register files. We have found that having more than 16 32-bit registers does not help. This is therefore what we assume here.

Software model

We examine two expansion modes, VPE First and the Basic Block Method. We use the appropriate memory layout for each, i.e. variables together for VPE First and VPEs (tiles) together for Basic Blocks.

Applications

1. BMAll – The low- and intermediate-level parts of ARPA IU Benchmark II [36]. This is an integrated series of tasks including convolutions, segmentation, convex hull, and many others. Substantial computation interspersed with communication. Both integer and floating point computations
2. motion256 – Depth from correspondence [14]. Contains substantial floating point computations with very little communication.
3. spiral3 – Correlation-based correspondence matcher. Dominated by communication. Integer computation only.
4. prewitt – A region-based edge-grouping line finder, based on the Burns algorithm [6]. Uses both a substantial number of integer operations and communications.

Results

We compare hit rates of VPE Expansion First (VEF) with Basic Blocks (BB). We vary application, VPR, and cache size. In the first series of graphs (Figure 7), we show BMAll for four different VPRs. In the second series (Figure 8), we show all four applications for a VPR of 64. In both cases, the per PE cache size ranges from 64 bytes to 8KB. The other hardware and software parameters are as described above. These results are representative of the several hundred thousand datapoints we have generated and whose detailed analysis will be the subject of a future study.

VPR captures the relationship between image size and array size: For example, a VPR of 64 represents an image size of 64K pixels and an array size of 1K PEs as well as an image size of 256K pixels and an array size of 4K PEs. Recall that the VPR also represents the maximum difference in working set size between VEF and BB.

The superiority of the basic blocks approach is clearly shown. In BMAll, cache must be roughly 4 times larger for VEF than BB to get good performance, i.e. a hit rate of over 90%. Also, BB approaches that performance more rapidly than VEF: BB achieves a 70% to 80% hit rate with very small caches even for large VPR, while VEF performance declines precipitously with a slight reduction in cache size.

In the second figure, we see the effects of application differences. BMAll and Prewitt are both comparatively

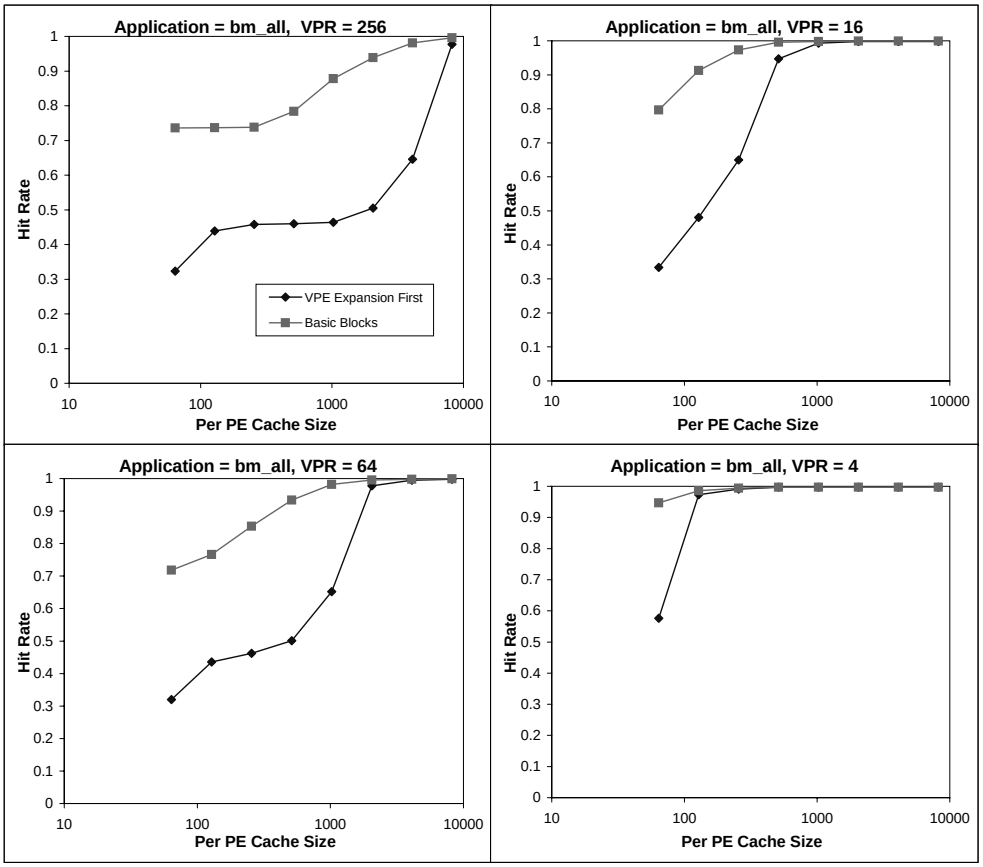

Figure 7: For BMAll, hit rate versus cache size while varying expansion method and VPR.

long codes with a number of different phases and a large variation in basic block sizes. Therefore, Prewitt and BMall both see a gradual increase in working set for both expansion methods, but with a more rapid increase for BB. Spiral3, although similarly computationally intensive to the other applications, is a very simple code with very small basic blocks. This explains the sharply defined working set and the simple tracking of performance of VEF and BB. We also see that this is an example of an effect described earlier: that applications with very frequent communication also tend to have small working sets. Motion256 on the other hand, has almost no communication, and therefore very large basic blocks, as well as the largest working set of the applications studied here. A 1K per PE cache is sufficient to achieve a 93% hit rate for BB, but even an 8KB cache is not enough to reach 90% for VEF.

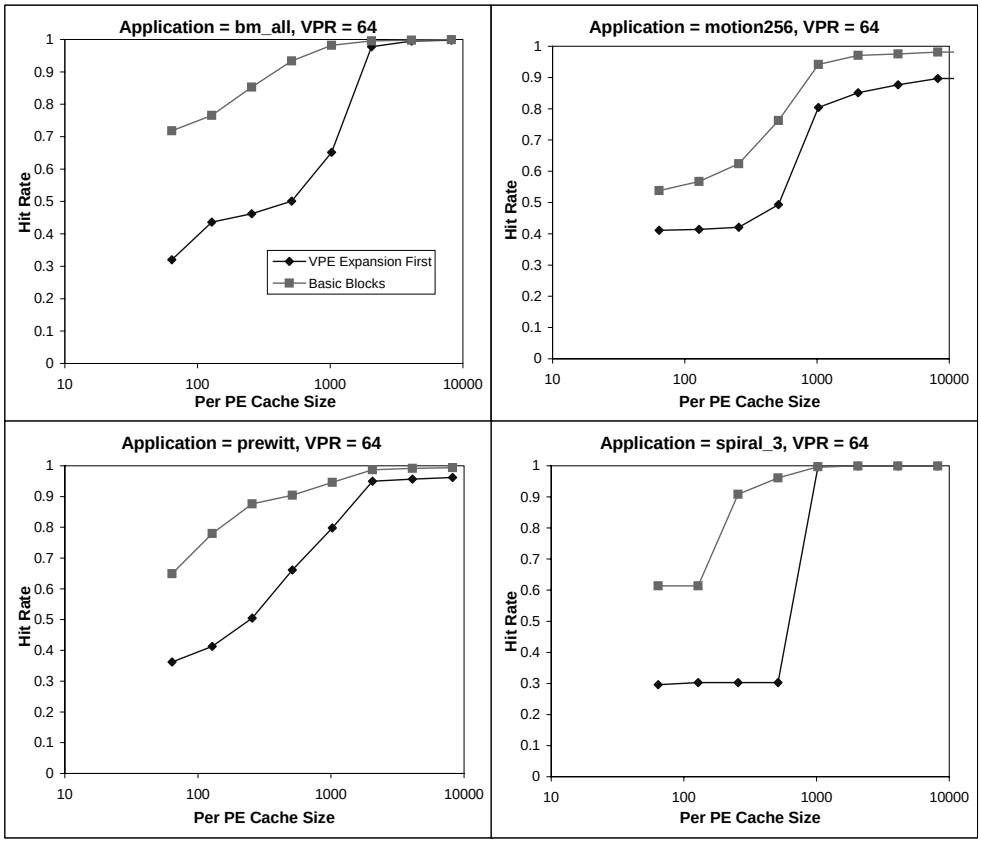

Figure 8: For VPR = 64, hit rate versus cache size while varying expansion method and application.

7 Conclusion

We have described the problems involved in instruction selection and issue for SIMD processor chips. Our solution involves preloading basic blocks of VPE macroinstructions and expanding them in two stages: by tile and by microcode. In this way we are able to keep up with high speed PEs and also maintain the locality in the PE array data stream. We have tested this approach on a number of image understanding applications using the ENPASSANT environment. We found that the basic block approach was superior in two ways. The first is that the cache size required to achieve a 90% hit rate is roughly four times smaller than was required in previous approaches. The other is that the performance degradation is more graceful as working set size increases: therefore, the performance is much more likely to be tolerable when the application is partially memory bound.

Another result is in characterizing the applications themselves and in the recommendation derived for on-chip

memory size. All of the applications have good performance with 32 bytes of on-chip memory per VPE (when using the basic block method). Therefore, for 8 bit pixels and this application set, an on-chip memory 32 times the image size is reasonable. We find this to be a very interesting result since it points to on-chip memory of from 2MB to 32MB. This is not only well within range of SRAM-based ASICs, but also is approached by that available in the current generation of high-end FPGAs, such as the Xilinx XC2VP100 [37]. Particularly significant is that these results are for production codes, not kernels, and that the code is not hand-crafted assembly language, but compiled with only automatic optimization from a high-level language.

Our solution does not lose effectiveness when applied to multiple ACU/PE chips: the ACU is simply replicated along with each PE chip. As interPE communication is more likely to become a bottleneck in these designs, many recent arrays have been one dimensional. Other solutions involve asynchronous transfers and other latency hiding techniques well known in multiprocessors.

Acknowledgments We thank the anonymous reviewers for their many helpful suggestions.

References

[1] Allen, J. D., and Schimmel, D. E. Issues in the design of high performance SIMD architectures. *IEEE Trans. on Parallel and Distributed Systems 7*, 8 (1996), 818–829.

[2] Bishop, B., Zhang, Y., Acken, K., Irwin, M., and Owens, R. Three dimensional graphics algorithms on the Micro-Grain Array Processor-II. In *Proc. of Computer Architectures for Machine Perception* (1997), pp. 204–208.

[3] Blank, T. The MasPar MP-1 architecture. In *Proc. 35th IEEE Comp. Conf.* (1990), pp. 20–24.

[4] Bolotski, M. *Abacus: A Reconfigurable Bit-Parallel Architecture for Early Vision*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1996.

[5] Broggi, A., and et al. A dedicated image processor exploiting both spatial and instruction-level parallelism. In *Proc. of Computer Architectures for Machine Perception* (1997), pp. 106–115.

[6] Burns, J. B., Hanson, A. R., and Riseman, E. M. Extracting straight lines. *IEEE Trans. on Pattern Analysis and Machine Intelligence PAMI-8*, 4 (1986), 425–455.

[7] Burrill, J. H. *Using the GNU C++ Compiler to Generate Code for a Massively Parallel SIMD Processor*. www.cs.umass.edu/ burrill/, 1993.

[8] Cantoni, V., and Guerra, C., Eds. *Proceedings of Computer Architectures for Machine Perception, CAMP '00*. IEEE Computer Society, Los Alamitos, CA, 2000.

[9] Cantoni, V., and Petrosino, A. 2-d object recognition by structured neural networks in a pyramidal architecture. In *Proc. Computer Architectures for Machine Perception* (2000), pp. 81–86.

[10] Chang, H., Ong, S., Lee, C., Sunwoo, M., and Cho, T. A general purpose Slim-II image processor. In *Proc. of Computer Architectures for Machine Perception '97* (1997), pp. 253–259.

[11] ClearSpeed Technology. MTAP processor core. Tech. Rep. 05-PO-1101 v2.4, ClearSpeed Technology, LTD, Hunts Ground Road; Stoke Gifford; Bristol, UK, 2002.

[12] DeFord, M. Test and integration environment for PCI coprocessor cards. Master's thesis, Department of Electrical and Computer Engineering, University of Houston, 2001.

[13] Dulac, D., Guezguez, S., and Bertrand, G. Parallel segmentation based on topology with the associative net model. In *Proc. of Computer Architecture for Machine Perception* (2000), pp. 95–103.

[14] Dutta, R. Parallel dense depth maps from motion on the image understanding architecture. In *Proc. of the 1993 IEEE Computer Society Conf. on Computer Vision and Pattern Recognition* (1993), pp. 154–159.

[15] Fountain, T. The design of highly-parallel image processing systems using nanoelectronic devices. In *Proc. of Computer Architectures for Machine Perception* (1997), pp. 210–219.

[16] Fujita, Y., Kyo, S., Yamashita, N., and Okazaki, S. A 10 GIPS SIMD processor for PC-based real-time vision applications. In *Proc. of Computer Architectures for Machine Perception* (1997), pp. 22–32.

[17] Fujita, Y., Yamashita, N., and Okazaki, S. A 64 parallel Integrated Memory Array Processor and a 30 GIPS real-time vision system. In *Proc. of Computer Architectures for Machine Perception* (1995), pp. 242–249.

[18] Gealow, J. C., Herrmann, F. P., Hsu, L. T., and Sodini, C. G. System design for pixel parallel image processing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 4* (1996).

[19] Grate, L., Diekhans, M., Dahle, D., and Hughey, R. Sequence analysis with the Kestrel SIMD parallel processor. In *Pacific Symposium on Biocomputing* (2001), pp. 323–334.

[20] Herbordt, M. C. *The Evaluation of Massively Parallel Array Architectures*. PhD thesis, Dept. of Comp. Sci., U. of Mass. (also TR95-07), 1994.

[21] Herbordt, M. C., Anand, A., Kidwai, O., Sam, R., and Weems, C. C. Processor/memory/array size tradeoffs in the design of SIMD arrays for a spatially mapped workload. In *Proc. of Computer Architectures for Machine Perception* (1997), pp. 12–21.

[22] Herbordt, M. C., Burrill, J. H., and Weems, C. C. Making a data-parallel language portable for massively parallel array computers. In *Proc. of Computer Architectures for Machine Perception* (1997), pp. 160–169.

[23] Herbordt, M. C., Cravy, J., Sam, R., Kidwai, O., and Lin, C. A system for evaluating performance and cost of SIMD array designs. *Journal of Parallel and Distributed Computing 60*, 2 (2000), 217–246.

[24] Herbordt, M. C., and Weems, C. C. Experimental analysis of some SIMD array memory hierarchies. In *Proc. 1995 Int. Conf. on Parallel Processing* (1995), vol. 1: Architecture, pp. 210–214.

[25] Hillis, W. D. *The Connection Machine*. The MIT Press, Cambridge, MA, 1985.

[26] Ishikawa, M., K.Ogawa, Komuro, T., and Ishii, I. A CMOS vision chip with SIMD processing element array for 1ms image processing. In *Proceedings IEEE Int. Solid-State Circuits Conference* (1999), pp. 206–207.

[27] Jonker, P., and Vogelbruch, J. The CC/IPP, an MIMD-SIMD architecture for image processing and pattern recognition. In *Proc. of Computer Architectures for Machine Perception* (1997), pp. 33–39.

[28] Jonker, P. P. *Morphological Image Processing: Architecture and VLSI Design*. Kluwer, The Netherlands, 1992.

[29] Komuro, T., Ishii, I., Ishikawa, M., and Yoshida, A. High speed target tracking vision chip. In *Proc. Computer Architectures for Machine Perception* (2000), pp. 48–56.

[30] Kyo, S., Koga, T., and Okazaki, S. IMAP-CE: A 51.2 GOPS video rate image processor with 128 VLIW processing elements. *NEC Research and Development 43*, 1 (2002).

[31] McConnell, R. Massively parallel SIMD computing on a single chip. In *Proc. of the 11th HOT Chips Symposium* (1999).

[32] Owens, R. M., Irwin, M. J., Nagendra, C., and Bajwa, R. S. Computer vision on the MGAP. In *Proc. Comp. Arch. for Machine Perception* (1993), pp. 337–341.

[33] Robinson, W., and Wills, D. Design of an integrated focal plane architecture for efficient image processing. In *Proc. Int. Conf. on Parallel and Distributed Computing Systems* (2002).

[34] Weems, C. C., Ed. *Proceedings of Computer Architectures for Machine Perception, CAMP '97*. IEEE Computer Society, Los Alamitos, CA, 1997.

[35] Weems, C. C., and Burrill, J. R. The Image Understanding Architecture and its programming environment. In *Parallel Architectures and Algorithms for Image Understanding*, V. Prasanna Kumar, Ed. Academic Press, Orlando, FL, 1991.

[36] Weems, C. C., Riseman, E. M., Hanson, A. R., and Rosenfeld, A. The DARPA image understanding benchmark for parallel computers. *JPDC 11* (1991).

[37] Xilinx, Inc. *Virtex-II Pro Platform FPGA User Guide*. San Jose, CA, 2002.