Scalable, Network-Connected, Reconfigurable, Hardware Accelerators for an Automatic-Target-Recognition Application

By

Wen-King Su   Ruth Sivilotti
Young Cho   Danny Cohen

With guidance and help from Sandia's

Brian Bray and Doug Doerfler

May 1998

The research described in this report was sponsored by the Defense Advanced Research Projects Agency. The earlier phases of the research, concentrating on the two-level-multicomputer architecture and hardware implementations, was sponsored under DARPA Order B861, and monitored by the USAF Rome Laboratory under contract number F30602-94-C-0270. The later phases of the research, concentrating on this ATR "challenge application," was sponsored under DARPA Order E258, and monitored by the US Army, Fort Huachuca, under contract number DABT63-96-C-0052.

Please address questions and comments about this report to <Cohen@myri.com>.

Myricom, Inc., 325 North Santa Anita Avenue, Arcadia, CA 91006   Tel: 626-821-5555
"http://www.myri.com"   Fax: 626-821-5316

Scalable, Network-Connected, Reconfigurable, Hardware Accelerators for an Automatic-Target-Recognition Application

Introduction

Image processing, specifically Automatic Target Recognition (ATR) in Synthetic Aperture Radar (SAR) imagery, is an application area that requires tremendous processing throughput. In this application, data comes from high-bandwidth sensors, and the processing is time-critical. There is limited space and power for processing the data in the sensor platforms or in battlefield ground stations. DoD's strong push for using commercial-off-the-shelf (COTS) technology, the very high non-recurring engineering (NRE) costs for low volume ASICs, and evolving algorithms limit the feasibility of using custom special purpose hardware. In addition, a scalable system is required as the different sensor platforms have different image pixel rates and different mission requirements have different target recognition throughput needs per pixel.

To meet this challenge, Myricom, under DARPA funding, has developed a compact scalable system for high-performance implementation of the SAR ATR algorithms that were developed by Sandia National Laboratories (SNL). This system is based on the use of multiple, concurrent, specially programmed, reconfigurable computing nodes, interconnected by Myrinet. To achieve high efficiency, through the exploitation of the unique characteristics of this algorithm (e.g., operations on single bits), special FPGA-based computing nodes were developed by Myricom and delivered to SNL.

In this report, we describe the ATR algorithm implementation using FPGA accelerators. We describe the ATR algorithm that was implemented, the implementation on a single FPGA, how the FPGA nodes are connected to make a scalable system, and report both simulated and measured performance.

This system is a two-level computing system whose first level provides the general purpose infrastructure of network interfacing, message handling, mapping, initialization, etc., by the LANai microprocessor. Its second level provides the application-specific computing, which in this case is performed by the Lucent ORCA FPGA.

The sections of this report describe:

1. Overview of the system
2. The computation task, as developed by Sandia National Laboratories
3. Simulations
4. Adapting the algorithm to the structure of the FPGA computing nodes
5. Data flow through the system
6. Software functionality
7. Hardware
8. Performance, present and future
9. Summary
10. Conclusions

1. Overview of the System

The general flow of information in the entire ATR system is:

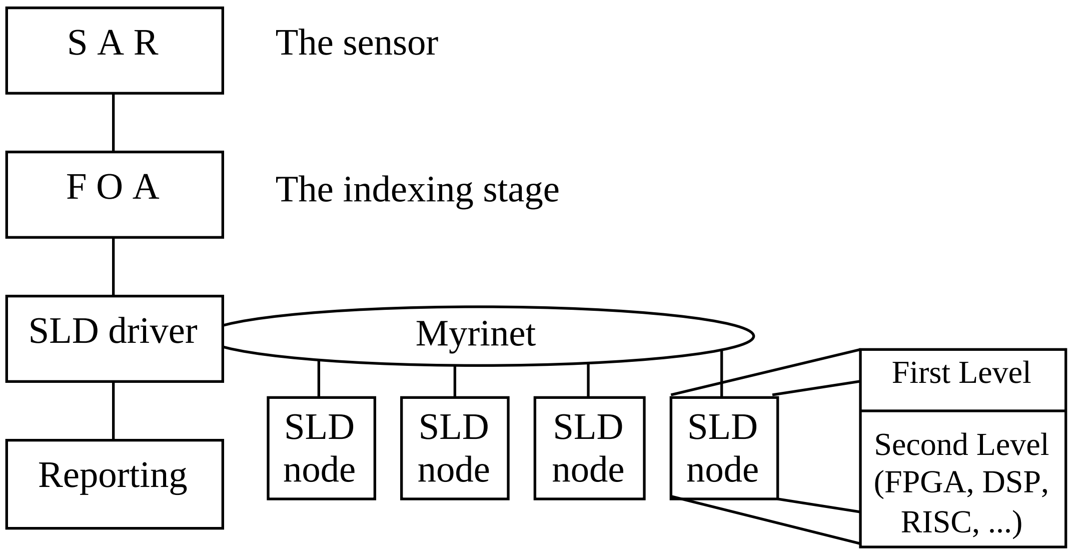

Figure 1: The overall ATR system

The FOA subsystem (for "Focus of Attention") handles the image as received by the SAR system (after its initial digital signal processing) and identifies locations where targets might be.

The SLD subsystem further checks these suspicious locations against target templates, by performing the SLD ("Second-Level Detection") operation. The SLD operation is "embarrassingly parallel", and could be performed by a multitude of various concurrent processing nodes which could be general purpose processors (such as Single-Board Computers, SBCs), or special purpose units.

The SLD driver then reports its best matches to the reporting section of the ATR system, which decides what to report according to various considerations.

The focus of the Myricom work is the SLD operation, which is checking ("correlating") image chips against target templates.

Sandia's current ATR systems are based on heterogeneous two-level multicomputers, microprocessors and DSP chips that are linked by Myrinet in a system area network (SAN) configuration providing a high level of fault tolerance, scalability, and load sharing. Myrinet is a scalable high-bandwidth network based on highly capable network interfaces and non-blocking crossbar switches. The first level of computing is the general network interface, and the second level of computing is the ATR specific programs running on microprocessors (such as a 200MHz Motorola 603ev PowerPC) and DSPs (such as a SHARC).

Myricom's implementation uses FPGAs for computing at the second level.

We have developed high-performance, FPGA-based, compact, reconfigurable computing nodes to perform the SLD tasks.

We have also developed a VME-6U baseboard with an 8-port crossbar chip (Xbar8) that connects 4 external Myrinet links with 4 connectors for mezzanine ("daughter") boards. These boards are the FPGA nodes described in this report.

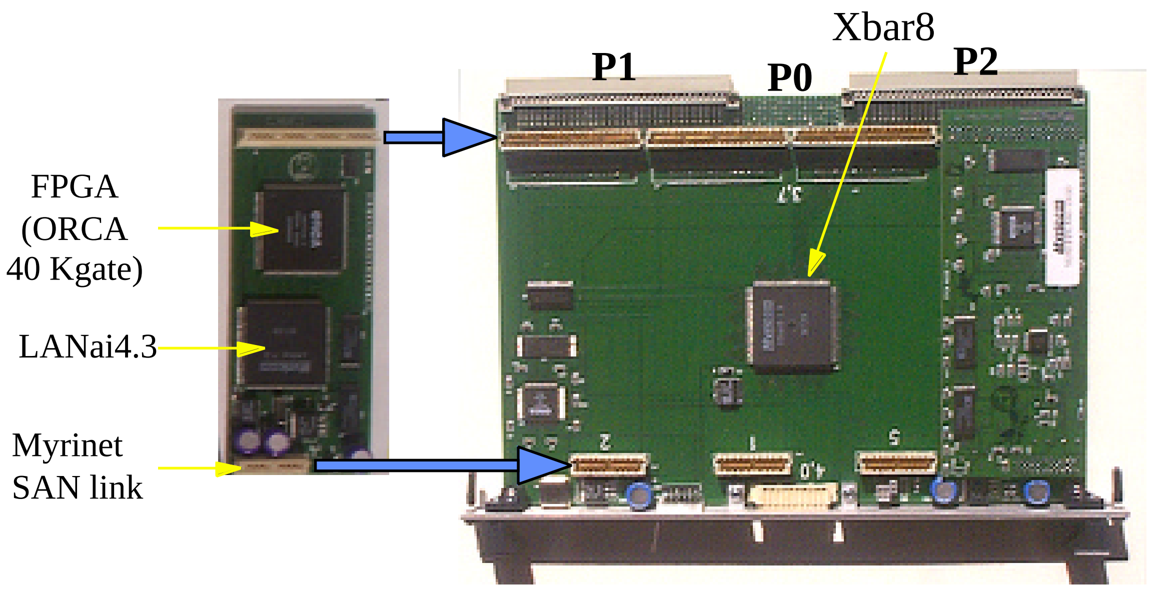

Figure 2: An FPGA mezzanine node (left) and a baseboard (right) with one node plugged to it

2. The Computation Task, as Developed by Sandia

Sandia National Laboratories' (SNL) real-time SAR ATR systems use a hierarchy of algorithms to reduce the processing demands for the SAR images, to yield a high probability of detection (Pd) and a low false alarm rate (FAR).

The first step in the SNL algorithm is a Focus of Attention (FOA) algorithm that runs over a down-sampled version of the entire image to find regions of interest that are of approximately the right size and brightness. The regions of interest are then extracted and processed by an indexing stage to further reduce the data stream which includes target hypotheses, orientation estimations, and target center locations. The surviving hypotheses have the full resolution data sent to an identification executive that schedules multiple identification algorithms and then fuses the results of the multiple identification algorithms.

The algorithm that we have implemented, using FPGA accelerators, is an indexing algorithm called Second-Level Detection (SLD). It is used for finding targets in-the-clear, not for camouflage, concealment or deception (CC&D) scenarios.

The SLD task is to take the extracted imagery (an image chip), match it against a list of provided target hypotheses, and return the hit information for each image chip consisting of the best two orientation matches, the degree of matching, the corresponding pixel location, and information about which target hypothesis gave rise to these two best matches.

SLD is a binary silhouette matcher that has a bright mask and a surround mask that are mutually exclusive. The bright mask and surround mask are 32x32 bitmaps, each having only about 100 non-zero pixels (about 10% of these bitmaps).

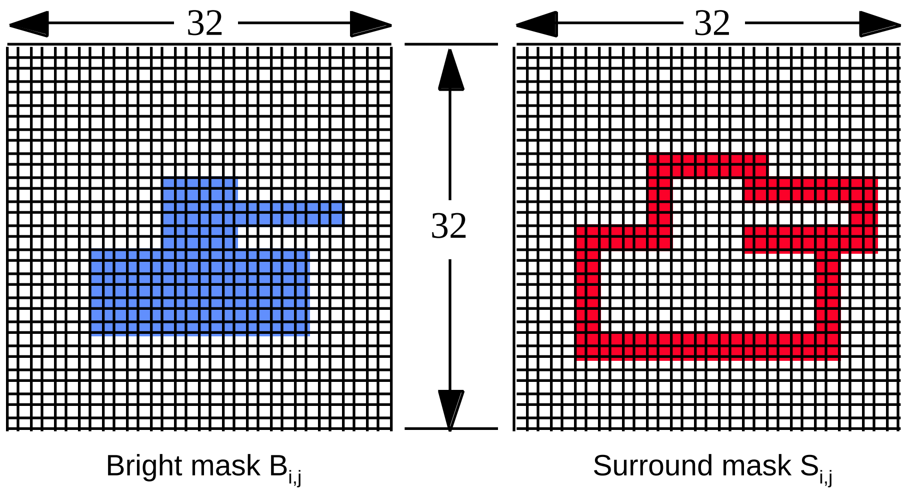

Figure 3: The two masks that are defined for every template

The system has a database of target models. For each target, and for each of a few elevations (typically 3) of the target, 72 templates are defined corresponding to all-around views of the target. The orientations of adjacent views are separated by 5 degrees.

Each template is composed of several parameters and two masks, a “bright mask” and a “surround mask”, where the former defines the image pixels that should be bright for a match, and the latter defines the ones that should not. These masks are mutually exclusive. Being “bright” is defined relative to a dynamic threshold (to be described later).

The FOA stage identifies interesting image chips, and composes a list of targets suspected to be in that chip. Having access to range and altitude information, the FOA algorithm also determines the elevation for that chip, without having to identify the target first.

The FOA tasks the SLD stage to evaluate the likelihood that the suspected targets are actually in the given image chip, and exactly where. To do so, the FOA defines tasks for the SLD stage, where each task is composed of an image chip, a suspected target with its elevation, one or two orientation intervals, and a few parameters.

Upon receiving these tasks, the SLD unit matches all the stored templates for this target and elevation and the applicable orientations with the image chip, and computes the level of matching (the “hit-quality”). The two hits with the highest quality are reported to the SLD driver as the most likely candidates to include targets. For each hit the template ID (specifying the target type, its orientation, and its elevation), the exact position of the hit in the search area, and the hit quality are provided, too.

After receiving this information the SLD driver reports this information to the ATR system.

The size of the bitmaps of the target-orientation templates is 32x32.

The size of the image chips is 64x64 8-bit deep pixels. The FOA algorithms guarantee that the target (if any) is located in the image chip such that a 6 pixel margin around the chip is guaranteed not to include the target.

Hence, the area of interest in an image is 52x52 pixels.

In a 52x52 area there are 21x21 possible places to position a 32x32 mask area (52-32+1=21). This defines a search-area of 21 search-rows, each 21 position wide, as illustrated in the following figure.

The position (i,j) in the search area corresponds to positioning the lower left corner of the mask over the pixel (i,j) of the image.

Hence, 2x21x21=882 matches (of an image and a mask) have to be performed for each orientation that is specified in the matching task.

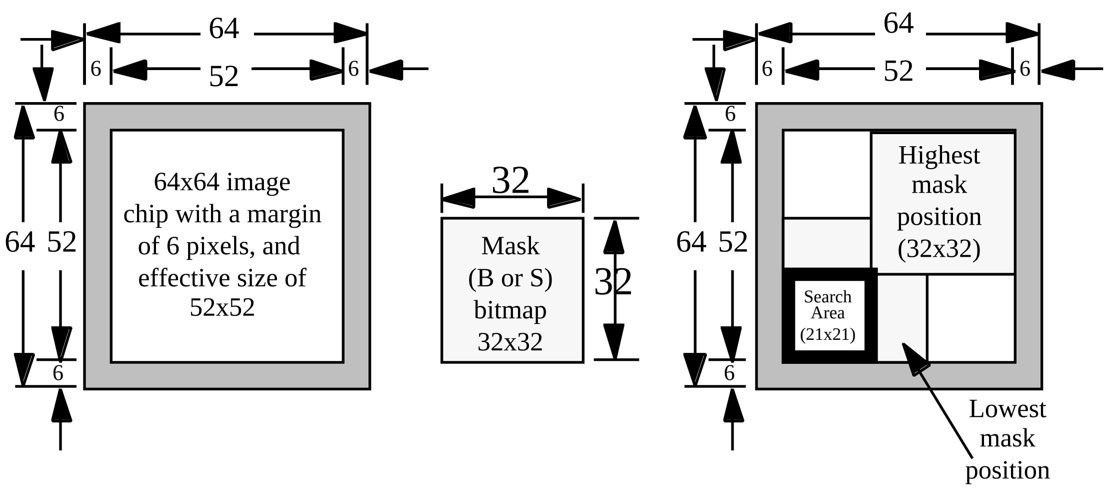
64
52
6
6
6
6
64
52
64x64 image chip with a margin of 6 pixels, and effective size of 52x52
32
32
Mask (B or S) bitmap 32x32
64
52
6
6
6
6
64
52
Highest mask position (32x32)
Search Area (21x21)
Lowest mask position

Figure 4: The relation between image chips, masks, and the search area

We use the following notation:

Image:	Image	M(a,b)	0...51 (or 6...57 inside of 0...63)
Templates:	Bright Mask	B(u,v)	0...31
	Bright Count	BC	
	Surround Mask	S(u,v)	0...31
	Surround Count	SC	
	Template bias	Bias	
	Minimal threshold	THmin	
	Maximal threshold	THmax	
	Minimal bright sum	BSmin	
	Minimal surround sum	SSmin	
Search Area:	Shape Sum	SM(i,j)	0...20
	Threshold	TH(i,j)	0...20
	Bright Sum	BS(i,j)	0...20
	Surround Sum	SS(i,j)	0...20
	Hit Quality	Q(i,j)	0...20
Performance:	(template/sec) per node	TSN	

The right column indicates the range for both variables.

The purpose of the first step in the SLD algorithm (called the “shape sum”) is to distinguish the target from its surrounding background. It consists of adaptively estimating the illumination (energy under the bright mask) for each position in the search area assuming that the target is at that orientation and location. If the energy is too little or too much then no further processing for that position for that template match is required. Hence, for each mask position in the search area, a specific threshold value is computed (not one threshold value for the entire image nor for the entire search area).

The purpose of the next step in the SLD algorithm is to roughly distinguish the target from the background by thresholding each image pixel with respect to the threshold of the current mask position, as computed before. The same pixel may be above the threshold for some mask positions, but below it for others.

This threshold calculation is to determine what is really a bright pixel and what is a surround pixel. The calculation consists of dividing the shape sum by the number of pixels in the bright mask (i.e., finding the average brightness under the bright mask) and subtracting a template specific constant (Bias).

The pixels under the bright mask that are ***greater than or equal to*** the threshold are counted, and if this count exceeds the minimal bright sum (BSmin) the processing continues. Now the pixels under the surround mask that are ***less than*** the threshold are counted. If this count exceeds the minimal surround sum (SSmin) it is declared a hit. The quality of the hit is the average of the percent of bright and surround pixels that were correct.

The 5 computing tasks {Pk, k=1...5}, for each position (i,j) in the search area (i,j=0...20), are:

P1: The shape sum at (i,j) is: $SM(i,j) = \sum_{u=0}^{31}\sum_{v=0}^{31} B(u,v)M(i+u, j+v)$

P2: The threshold at (i,j) is: $TH(i,j) = \frac{SM(i,j)}{BC} - Bias$

where *BC* is the number of 1's in the bright mask, and *Bias* is a template-specific value.

P3: The bright sum at (i,j): $BS(i,j) = \sum_{u=0}^{31}\sum_{v=0}^{31} B(u,v)\left[\; M(i+u, j+v) \geq TH(i,j) \;\right]$

where [TRUE]=1 and [FALSE]=0.

P4: The surround sum at (i,j): $SS(i,j) = \sum_{u=0}^{31}\sum_{v=0}^{31} S(u,v)\left[\; M(i+u, j+v) < TH(i,j) \;\right]$

A valid hit is at (i,j) if the 4 following conditions hold:

$$\begin{aligned} TH(i,j) &\leq TH\max \\ TH(i,j) &\geq TH\min \\ BS(i,j) &\geq BS\min \\ SS(i,j) &\geq SS\min \end{aligned}$$

P5: Finding and reporting the 2 valid hits with the highest hit quality.

The hit quality is defined by $Q(i,j) = \frac{1}{2}\left(\frac{BS(i,j)}{BC} + \frac{SS(i,j)}{SC}\right)$

where SC is the number of 1's in the surround mask.

3. Simulations

In order to verify our understanding of the SNL algorithm, we first implemented it in C, and ran it on a sample data set that we received from SNL. Our simulations reproduced the expected results received from SNL.

Over time this algorithm simulator has evolved into a full hardware simulator and verifier. It also allowed us to investigate various tradeoffs without actually implementing them in hardware.

This data set from SNL includes 2 targets, each with 72 templates, for 5-degree orientation intervals. Hence, in total we have 144 bright masks and 144 surround masks, each a 32x32 bitmap. The data set also includes 16 image chips (each 64x64 pixels at 1 byte/pixel).

Given a template and an image, there are 21x21=441 matrix correlations that must take place for each mask. This corresponds to 21 search rows, each 21 positions wide. The total number of search-rows which we correlate is:
(2*72 templates)*(16 images)*(21 (search-rows/image)/template)=48384 search-rows.

By analyzing the results of the simulations we have learned several important lessons:

Lesson-1: Check SS before BS

Using the given data the simulation yields:

Bad TH search-rows:	28841	(60%)
Bad SS search-rows:	19474	(40%)
Bad BS search-rows:	2	(0%)
Good search-rows:	67	(0%)
Total search-rows:	48384	(100%)

A bad-TH row is a row where for each j: (TH(i,j)>THmax) or (TH(i,j)<THmin).

A bad-SS row is a row that is not a bad-TH row, and where for all j: SS(i,j)<SSmin

A bad-BS row is a row that is neither a bad-TH row, nor a bad-SS row,
and where for all j: BS(i,j)<BSmin

Therefore it’s better to check for bad-SS (i.e., if for all j: SS(i,j)<SSmin) before checking for bad-BS (i.e., if for all j: BS(i,j)<BSmin).

The low rejection rate by (BS(i,j)<BSmin) is the result of TH being computed using only the B-mask, regardless of the S-mask. TH(i,j) is computed exactly by the same pixels that are used for computing BS(i,j).

In this data set, only 67 search-rows passed all these 4 conditions (as entire rows), and out of their 67*21=1407 positions, only 84 passed them (as individual positions), and were declared as hits.

Lesson-2: Skip zero mask rows

Each masks has 32 rows. However, many masks have all-zero rows which can be skipped. By storing with each template a pointer to its first non-zero row we can skip directly to the first non-zero row "for free". Embedded all-zero rows are also skipped. But the computation to find them is hidden in the pipeline.

The simulation tools showed that, for the template set that we have received, this optimization significantly reduces the range for u (see figure below).

If the mask is B(u,v) then the range of u is its "length", and the range of v is its "width".

This skipping works best on "short" masks (i.e., masks with a small "length"), such as on the mask on the left in the following figure.

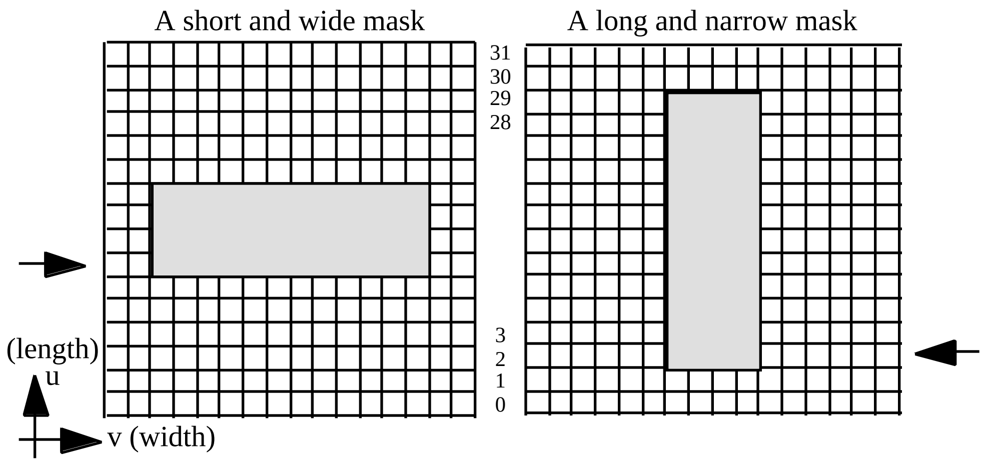

Figure 5: Short (left) and long (right) masks.

The data set has 72 template/target for 2 targets, for a total of 72*2=144 templates. Each template has 32 rows (of both masks) for a total of 144*32=4608 rows. Out of these 4608 bright-rows and 4608 surround-rows there are only 2206 non-zero bright-rows and 2815 non-zero surround-rows.

As expected, there are less non-zero bright-rows than surround-rows because the bright mask defines the body of the target, and the surround mask defines the external background around the target. Since the bright mask is used twice (for both TH and BS) whereas the surround mask only once (for SS) skipping the zero rows reduces the number of row operations from 4608+4608+4608=13824 to 2206+2206+2815=7227, which is only 52.3%, a saving of 1.9X.

The number of “internal” zero rows of the templates, according to our data, is very small, only 146 internal zero rows out of 9216 mask rows.

Lesson-3: : Skip zero mask columns

Just as Lesson-2 reduces the range for u (along the “length” of the masks) it is possible also to reduce the range of v (along the “width” of the masks) by skipping zero columns.

As seen later, our FPGA implementation works on an entire search-row concurrently. Hence, skipping rows reduces time, but skipping columns reduces the number of active elements that work in parallel, yielding no saving. Therefore, Lesson-2 is beneficial to our implementation, but Lesson-3 is not. We benefit from short masks (regardless of their width) but not from long masks.

Lesson-4: Transposing narrow masks

In order to benefit from narrow masks (as suggested in Lesson-3) it is possible to transpose narrow masks and correlate them against transposed images, allowing the same time savings for narrow masks as could be achieved for short masks (without rotation).

Hence, narrow masks (i.e., with a width that is smaller than their length) could be stored rotated, and correlated with rotated images, with the benefits of Lesson-2. If the number of templates that are to be matched with the image is large, the rotation task is justified.

4. Adapting the Algorithm to the Structure of the FPGA Computing Nodes

Each computing node has a LANai4 RISC processor, 512KB SRAM, and an FPGA (ORCA, 40K gate), as shown in the following figure.

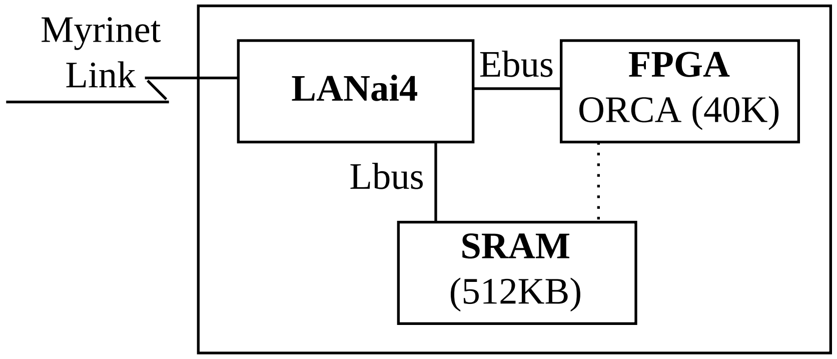

Figure 6: The functional structure of the FPGA node

When we designed the system, FPGA devices were optimized to support quick prototyping. FPGAs provide an advantage over custom chips and ASICs, because they can be modified quickly and at low cost after leaving the fabrication vendor.

These FPGA devices are not well suited to realtime reconfiguration, even though it is possible to reconfigure them at run time. The time required to reconfigure an FPGA is a dead time during which no computational progress may be achieved. Hence, minimizing reconfiguration time during computation is a good guideline for effective FPGA use.

Incidentally, most Myricom products contain FPGA devices which are configured once in their life time, before being shipped from Myricom to their users.

Another guideline for effective use of FPGAs is to concurrently perform as many operations as possible.

The use of FPGAs as compute engines allows the hardware to take on a large range of task parameters through reconfiguration.

All three FPGA computing tasks (P1, P3, and P4) correlate a sliding mask with image data (possibly quantized by a threshold). They all need some variation of:

$$G(i,j) = \sum_{u=0}^{31} \sum_{v=0}^{31} F\big(B(u,v),\ M(i+u, j+v) \big)$$

Where G(i,j) is SM(i,j), BS(i,j) or SS(i,j), and B(u,v) is a bit from either mask.

Our design uses 21 units, {U(j), for j=0,...,20}, such that G(i,j) is computed by U(j). Hence, j is spread ***in space*** (concurrently, 21 times) while i, u, and v are spread ***in time*** (sequentially) in 21*32*32 cycles..

This scheme covers the entire 21x21 search area, by computing each search-row (21 pixels wide) in parallel.

A simple example may help in understanding the operation. For simplicity, in this example the image data is of size 6x6 (instead of 52x52) where a and b range over 0...5. The mask size is 3x3 (instead of 32x32) where u and v range over 0...2. The search area is 4x4 (instead of 21x21), because 4=6-3+1 (similar to 21=52-32+1), where i and j range over 0…3.

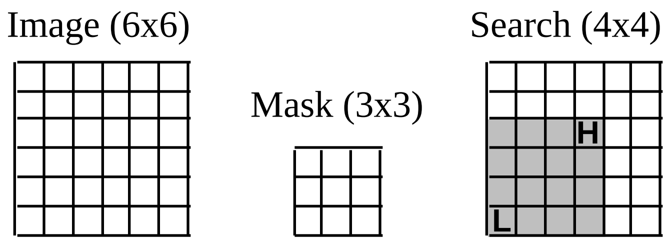

Figure 7: The area sizes for the given example

The P1 computing task is to compute for (i,j=0...3):

```
SM(i,j)= B(0,0)*M(i+0,j+0) + B(0,1)*M(i+0,j+1) + B(0,2)*M(i+0,j+2) +
         B(1,0)*M(i+1,j+0) + B(1,1)*M(i+1,j+1) + B(1,2)*M(i+1,j+2) +
         B(2,0)*M(i+2,j+0) + B(2,1)*M(i+2,j+1) + B(2,2)*M(i+2,j+2)
```

At the first "i-sequence", for i=0, U0 computes SM00, U1 computes SM01, U2 computes SM02, and U3 computes SM03.

Let's write specifically the expressions for SM(0,j), j=0...3, using a shorthand notation:

```
U0: SM00=B00*M00+B01*M01+B02*M02+B10*M10+B11*M11+B12*M12+B20*M20+B21*M21+B22*M22
U1: SM01=B00*M01+B01*M02+B02*M03+B10*M11+B11*M12+B12*M13+B20*M21+B21*M22+B22*M23
U2: SM02=B00*M02+B01*M03+B02*M04+B10*M12+B11*M13+B12*M14+B20*M22+B21*M23+B22*M24
U3: SM03=B00*M03+B01*M04+B02*M05+B10*M13+B11*M14+B12*M15+B20*M23+B21*M24+B22*M25
```

Each of these expressions has 9 terms, each is evaluated and accumulated in 9 successive cycles as shown below:

```
Cycle:     0       1       2       3       4       5       6       7       8
 i:        0       0       0       0       0       0       0       0       0
 u:        0       0       0       1       1       1       2       2       2
 v:        0       1       2       0       1       2       0       1       2
U0: SM00=B00*M00+B01*M01+B02*M02+B10*M10+B11*M11+B12*M12+B20*M20+B21*M21+B22*M22
U1: SM01=B00*M01+B01*M02+B02*M03+B10*M11+B11*M12+B12*M13+B20*M21+B21*M22+B22*M23
U2: SM02=B00*M02+B01*M03+B02*M04+B10*M12+B11*M13+B12*M14+B20*M22+B21*M23+B22*M24
U3: SM03=B00*M03+B01*M04+B02*M05+B10*M13+B11*M14+B12*M15+B20*M23+B21*M24+B22*M25
```

When the above sequence ends, it is repeated for i=1, then i=2, and i=3. v is nested in u, nested in i, but parallel in j. Hence, at the second "sequence", when i=1, U0 computes SM10, U1 computes SM11, U2 computes SM12, and U3 computes SM13.

There are several important things to notice:

First, all the units get the same B(u,v) at the same time. This suggests broadcasting the B(u,v) coefficients to all the Uj units.

Second, the image data that is used by unit U(j) at any cycle is used by the unit U(j-1) at the next cycle (except when v returns to 0). This suggests pipelining the pixels, M, through the Uj units.

When u changes, a new set of image pixels has to be processed. This set is the next image row, except when u returns to 0.

In general, when u changes, the (i+u)th row of pixels is used.

The computing structure for implementing the above is shown in the figure below:

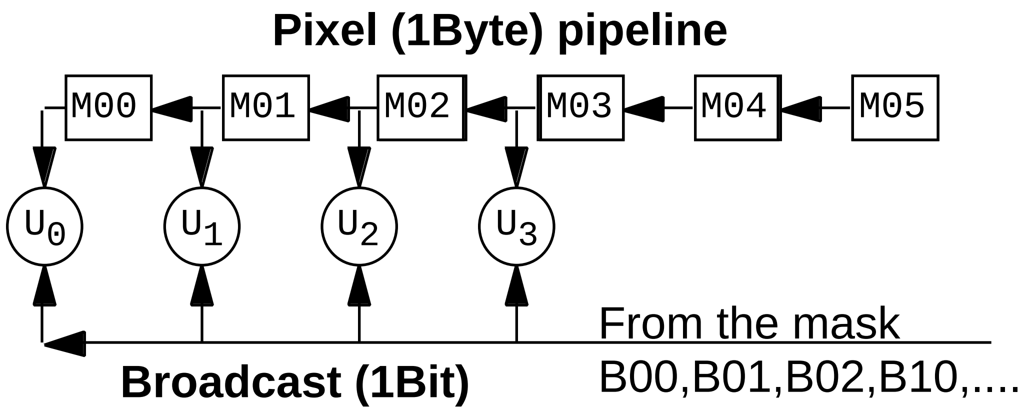

Figure 8: The general structure with pixel pipeline and mask broadcast

In order not to waste time while changing the rows of pixels, the pixels pipeline has the capability either to operate as a pipeline (aka a FIFO) or to be directly loaded from another set of registers.

At every clock cycle each Uj unit performs one operation, v is incremented modulo 3, and the pixel pipeline shifts by one stage (U1 to U0, U2 to U1, ...). When v returns to 0, u is incremented modulo 3, and the pixel pipeline is loaded with the entire (i+u)th row of the image.

When u returns to 0, the results are offloaded from the Uj (using another pipeline that is not shown here), their accumulators are cleared, and i is incremented modulo 4. When i returns to 0, this computing task (P1, P3, or P4) is completed.

The initial loading of the pixel pipeline (with an image row) is from the image-word pipeline that is word wide, hence 4 times faster than the image-pixel pipeline. This speed advantage guarantees that it will be ready with the next image row when u returns to 0.

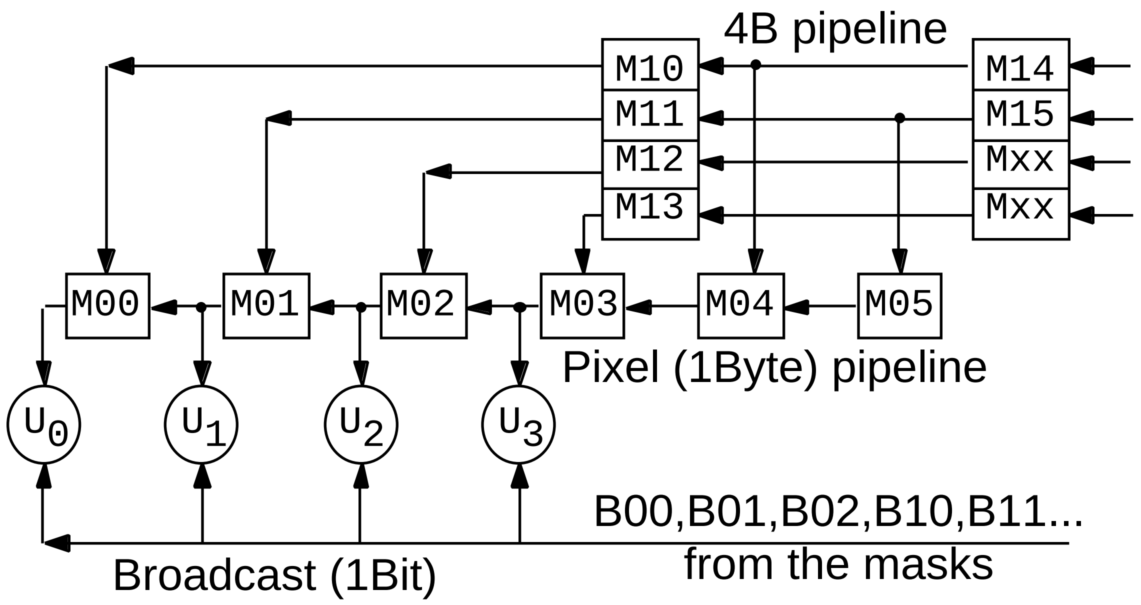

Figure 9: The word-pipeline for initializing the pixel-pipeline

The three computing tasks needed for stages P1, P3, and P4 are:

For P1: Uj has to compute: $SM(i,j) = \sum_{u=0}^{31}\sum_{v=0}^{31} B(u,v)M(i+u,j+v)$

This is accomplished by:

When u returns to 0: Uj = 0
in every other cycle: if (B) then Uj += M
where: B is the bit from the bright mask, being broadcasted,
M is the pixel value available from the pixel pipeline.

The following figure shows the computing structure for P1.

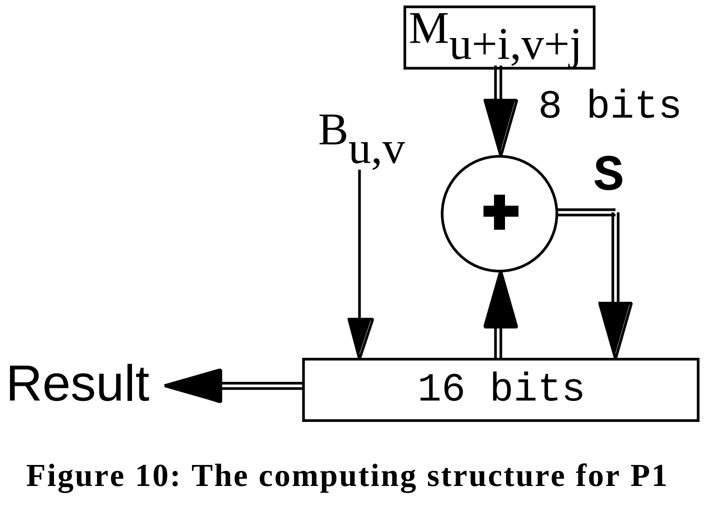

Figure 10: The computing structure for P1

For P3: Uj has to compute $BS(i,j) = \sum_{u=0}^{31}\sum_{v=0}^{31} B(u,v)\left[\ M(i+u, j+v) \geq TH(i,j)\ \right]$:

This is accomplished by:

When u returns to 0: Uj = 0
in every other cycle: if (B) then if (M≥THj) then Uj++
where B is the bit from the bright mask, being broadcasted,
M is the pixel value available from the pixel pipeline

THj is TH(i,j), as computed by P2.

The following figure shows the computing structure for P3.

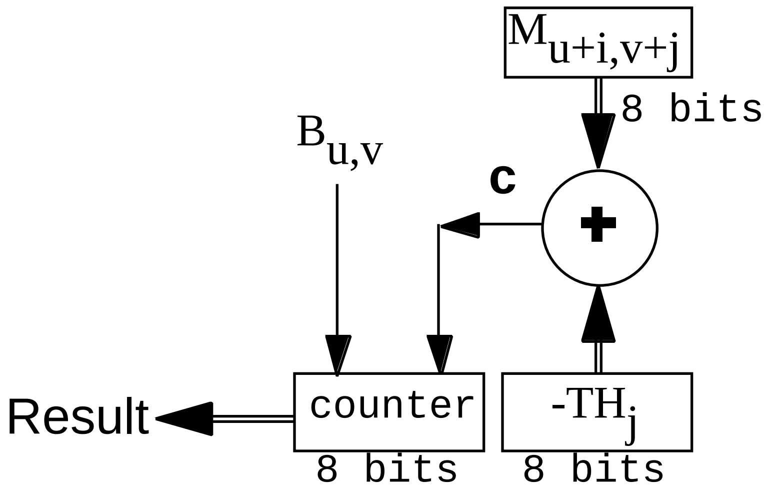

Figure 11: The computing structure for P3

For P4: Uj has to compute: $SS(i,j) = \sum_{u=0}^{31}\sum_{v=0}^{31} S(u,v)\left[\ M(i+u, j+v) < TH(i,j)\ \right]$

This is accomplished by:

When u returns to 0: Uj = 0
in every other cycle: if (B) then if (M<THj) then Uj++
where: B is the bit from the surround mask, being broadcasted,
M is the pixel value available from the pixel pipeline

THj is TH(i,j), as computed by P2.

The following figure shows the computing structure for P4.

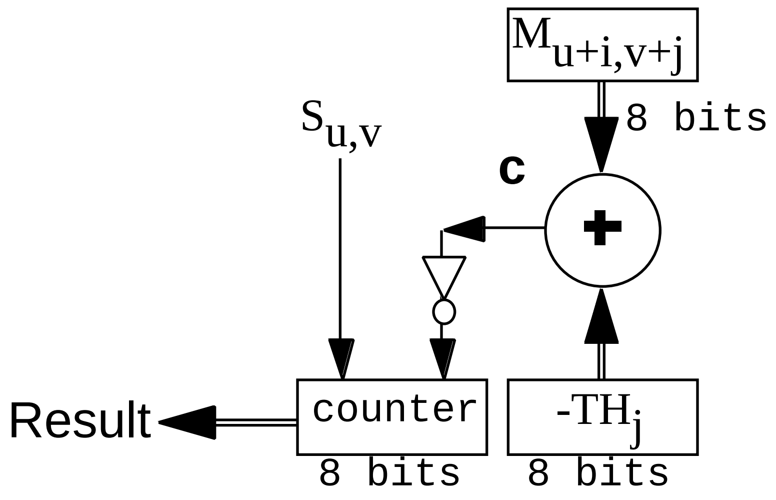

Figure 12: The computing structure for P4

Developing different efficient FPGA programs (or “structures”) for P1, P3, and P4 is an interesting approach to solving this problem. At the end of each stage the FPGA device would be reprogrammed (or “reconfigured”) with the optimal structure for the next task.

Using this approach matching templates with images at 1KHz requires reconfigurations at 3 KHz, with each reconfiguration limited to 333 μsec (or less if any computing task has to be performed in addition to the reconfigurations). As appealing as this approach may sound, it is not very practical since currently available FPGA devices have typical reconfiguration times of hundreds of milliseconds.

Therefore, we resisted the temptation to perform dynamic reconfigurations. Instead, we designed one structure to perform all three stages.

In all three stages there is a need to bring in pipelined pixels (M), according to the same sequence (v in u in i), to broadcast one bit (B) from a mask, and to modify the value of Uj as a function of a certain pixel (M), and B.

In P1 the modification of Uj is Uj+=M which is an addition of the 8-bit pixel value M, to the 16-bit number Uj. In P3/P4 it’s Uj++, where Uj is an 8-bit number.

In P1 the condition for this modification is the value of the B bit. In P3/P4 the condition is, in addition to the B bit, also the comparison of M with THj. Since THj does not depend on u and v, it’s a constant until the next change of i.

The modification of Uj is simpler in P3/P4 than in P1, but the evaluation of the condition for modification is simpler in P1 than in P3/P4.

This suggested that the total complexity is about the same in P1 and in P3/P4 and suggested a path to commonality.

An important observation was that adding an 8-bit number to a 16-bit number is the same as adding two 8-bit numbers, M and UjL (L for “low”), and conditionally incrementing UjH (H for “high”) if the addition overflows.

This suggests that each stage needs a conditional 8-bit counter, and an 8-bit adder for either adding the pixel M to UjL or for comparing M with THj. This operation is conditional on the mask (B=1).

Both P3 and P4 evaluate exactly the same condition -- comparing M with THj (except that P3 counts if the overflow is 0 and the P4 if it's 1). Therefore they can share a single comparison. This suggests that we perform both P3 and P4 at the same time. We call this combined operation P34.

The FPGA real estate required to perform both P3 and P4 at the same time is less than twice the real estate required for each separately. They share the distribution of M and THj, and the 8-bit adder, but each requires its own broadcast of B (because P3 needs the bright mask whereas P4 needs the surround mask), and each requires its own 8-bit counter.

Hence, the structure of Uj (see following figure) requires an 8-bit adder with the inputs M and R0, whose result is conditionally loaded into R0 and two 8-bit counters, R1 and R2.

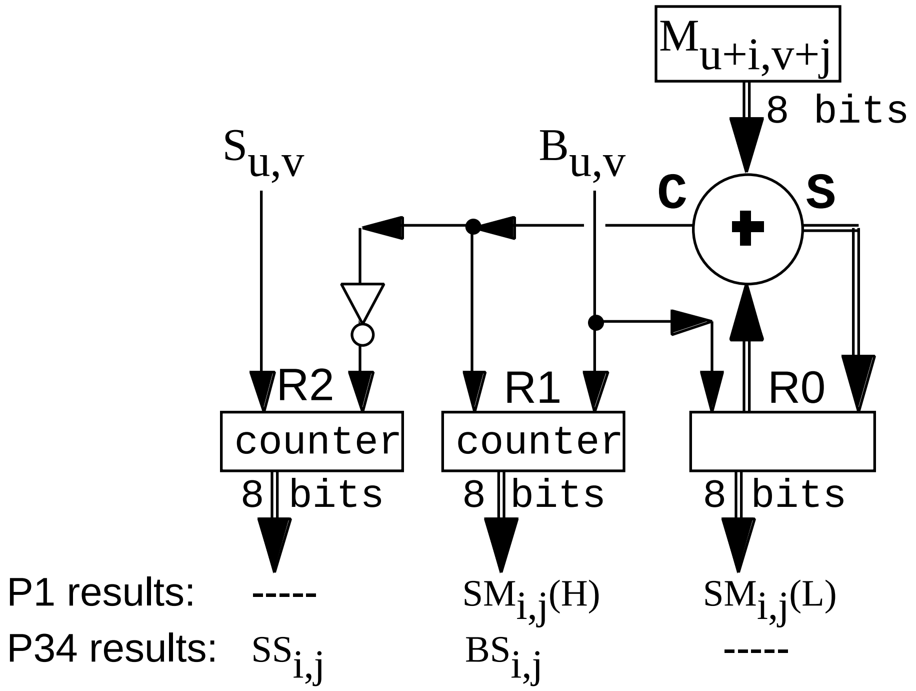

Figure 13: The structure for computing both P1 and P34

The use of the 8-bit registers:

Phase:	P1	P34
R0-register:	UjL	-TH(i,j)
R1-counter:	UjH	BS(i,j)
R2-counter:	------	SS(i,j)

Conditions for update:

In P1: increment R1 if (B=1) AND (the 8-bit addition of M to R0 overflows).
In P34: increment R1 if (B=1) AND (the 8-bit addition of M to R0 overflows)
increment R2 if (S=1) AND (the 8-bit addition of M to R0 does not overflow)

The results of P1 are (R0,R1) and of P34 are (R1,R2).

This approach reduces the number of cycles needed to perform P1, P3, and P4 from 3*32*32*21*21 operations to 2*32*32*21 (31.5X improvement).

Above, we described in detail the tasks P1, P3, and P4 that are performed by the FPGA. There are two additional computing tasks, P2 and P5, that are performed by the LANai on the computing node.

P2 computes the threshold, TH(i,j), by dividing the shape sum, SM(i,j), that was computed by the FPGA, by a constant (BC, the number of 1's in the bright mask) and subtracting another constant (the Bias).

The values of Bias, BC, SC, THmax, THmin, BSmin, and SSmin are contained in the templates.

The total number of additions for P1, P3, and P4 is 3*32*32*21*21. P2 uses only 21*21 divide+add operations.

Since the duty cycle of P2 is only 1/(3*32*32) of the entire FPGA operation, our first approach was ***not*** to dedicate any FPGA real estate to this division and to have it performed by the LANai.

After the FPGA completes the task P34, the LANai has to perform task P5, which includes finding the two matches with the highest hit quality, and forwarding the results, according to the architecture that is used (as described in the following section "The data flow through the system").

The hit quality is defined by Q(i,j) = (BS(i,j)/BC + SS(i,j)/SC) / 2 where SC is the number of 1's in the surround mask.

Since typically BC and SC are similar in value (about 100) the LANai on the computing node estimates Q(i,j) by BS(i,j)+SS(i,j) for ordering purposes, and leaves the exact computation of Q(i,j) to the software that runs on a more capable system.

5. Data Flow through the System

The core of the SLD task is to compare an image with a set of templates to find a best correlation, or match. We have seen that the FPGA can effectively perform the computations that are necessary to correlate a single image with a single template. The next question is: how best to partition the task of correlating many images with many templates over an array of FPGA nodes.

In this section we discuss several data flow topologies that may be implemented on top of an arbitrary physical Myrinet topology.

We have implemented three methods for distributing this task over the nodes. The biggest difference between these designs is how messages flow through the system and what these messages consist of. We will refer to these designs as architectures A1, A2 and A3. These are just a few of the myriad topologies that could be used.

5.1 Design A1

In A1, each FPGA node works independently (of the other nodes) to find matches. Each node contains a complete set of templates. The host will give each node a match task (consisting of an image and ranges of templates to correlate the image with), and the node will find the best two matches and return the answer to the host. The host will keep the nodes well supplied with match tasks.

The communications resemble spokes emanating from a central hub (see the figure below) - note that the spokes represent lines of bi-directional communication (match tasks from the host to the nodes, best matches from the nodes to the host). There is no communication between the FPGA nodes, only between the nodes and the host.

This architecture is fault tolerant with respect to node failures - if the host does not hear from a node it can easily resend the match task to another node, and continue to find matches with the remaining nodes. Also, the FPGA nodes can be kept well supplied with tasks - as each node finishes a task it can be given another - resulting in a high utilization of the compute nodes.

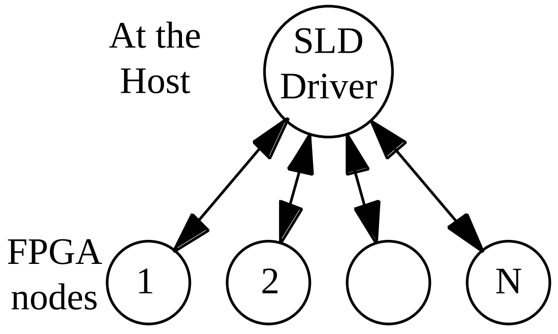

Figure 14: A1, communication is between the host and each node.

In this architecture each node stores all the templates.

5.2 Design A2

A2 partitions the task by distributing the templates over the nodes which are arranged in a chain. Images and their current best matches are passed down the chain from node to node as each node finishes matching an image against the node's subset of the templates. The two best matches so far are passed along with the image and then returned to the host by the last node.

A2 is shown in the following figure, where arrows indicate the direction of data flow. Flow control messages travel in the opposite direction.

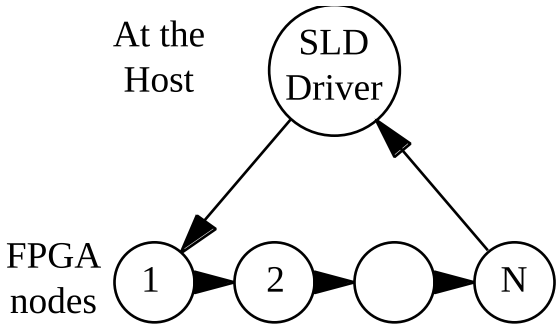

Figure 15: A2, communication along a ring

The data that is being passed is the same throughout the ring; it contains an image, the match specifications (for example, which templates should be correlated with the image) and a description of the two best matches so far. Because match tasks are likely to require that images be compared against consecutive templates, and we wish to balance the computation task across the nodes, we would distribute the templates so that node i out of N nodes has templates (i, i+N, i+2N...).

Each node would store about T/N templates, where N is the number of nodes and T is the total number of templates.

The advantage of A2 over A1 is that it is more scalable with respect to the number of templates. If the template set is very large, each node is A1 may not be able to store all the templates. Also, if the number of nodes in the system is quite large, A2 requires simpler bookkeeping on the host (since all messages come from one node, and the match tasks are naturally queued in the system). However, in A2 if one node or link is lost or becomes faulty, the nodes must re-route the match tasks, redistribute the templates of the "lost" node and then check the image against these templates. Dynamic load balancing is not optimal because, even if the same number of templates have to be matched by each node, the amount of work that each node has to do varies, and a node that is momentarily overwhelmed may create idle nodes downstream.

5.3 Design A3

A3 combines features from A1 and A2 (see figure below). The host may communicate with some rings of nodes (of various sizes) as well as some independent nodes (equivalent to rings of size 1). This hybrid approach would combine the best of both architectures and allow users to customize their system to fit their needs. For example, if the user had different classes of templates (one for matching tanks, another for planes, etc.) then creating a ring of nodes for each class would enable the algorithm to quickly match an image against all templates of one class. Also, using multiple rings would provide better fault tolerance than a single ring - if one node is lost, only one ring (and some percentage of the total nodes) will be affected.

An example of A3 is shown in the following figure, where FPGA nodes 1 through n are in one ring, nodes (n+1) through (N-1) are in another ring, and node N operates independently.

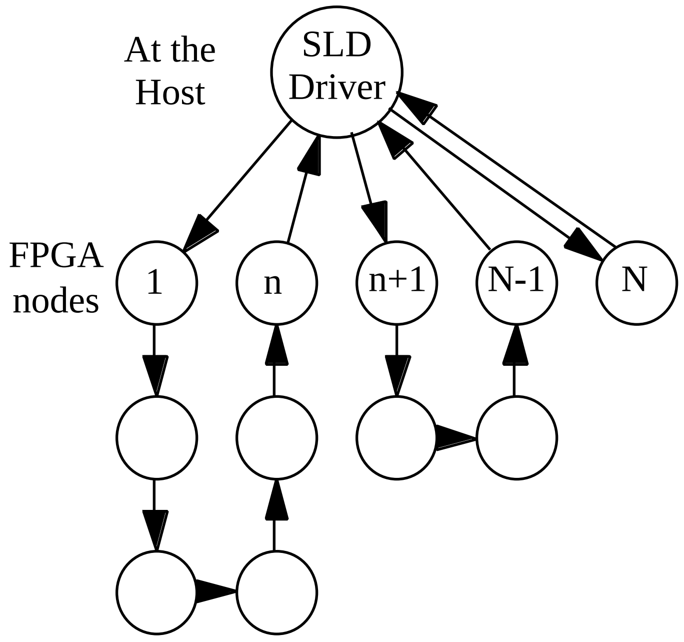

Figure 16: A3, with three rings

Another topology is shown below. It is a variant of A3, with several identical rings. It has 3 rings that are used in parallel, arranged as a 4-stage trellis to increase their fault tolerance, and to improve the dynamic load balancing.

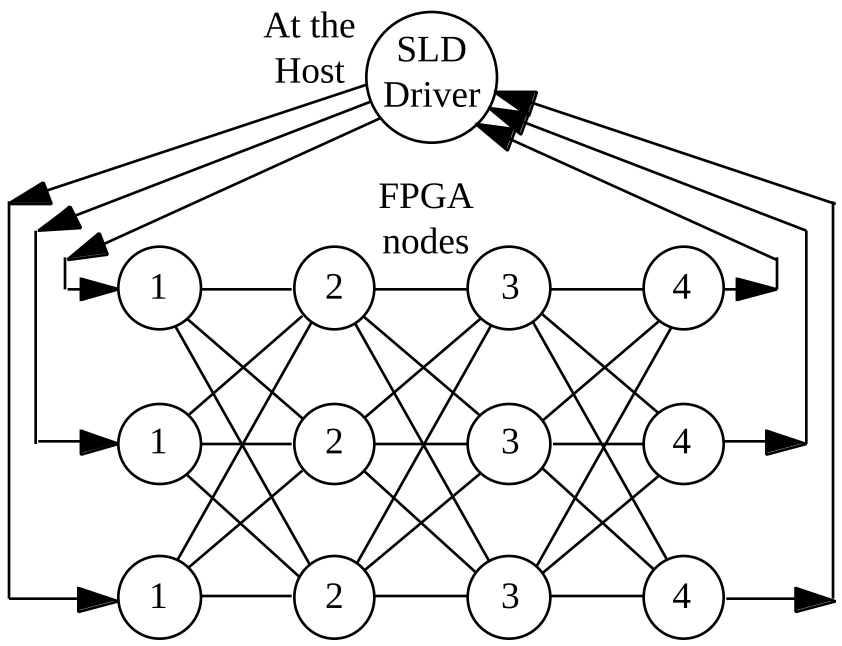

Figure 17: A trellis consisting of 3 parallel identical rings

All nodes in a column, have the same templates set. Upstream flow control messages are used.

5.4 Example

We include an example which illustrates how the optimal number of nodes per ring may be determined. In this example we consider having only one target type, such that all rings must have all the templates.

M match/sec are required. Each node can perform only m match/sec.

T templates are required. Each node can store only t templates.

The number of nodes must be greater than both NP=M/m and NS=T/t ("P" for processing and "S" for storage).

The total number of nodes in the ***system*** must be at least NP.
The total number of nodes in every ***ring*** must be at least NS.

R rings may be used, where R≤int(N/NS) with at least NS nodes in each ring.

For example, let us suppose that there are 23 nodes. 1000 match/sec are needed. Each node can do only 100 match/sec. 200 templates are needed. Each node can store only 40.

NP = 1000/100 = 10, and NS = 200/40 = 5.

N is greater than both NP and NS. (Hence, we can meet the requirements!)

Up to R = 23/5 = 4 rings may be used.

If 4 rings are used, each may have at least 23/4=5 nodes (e.g., 1 ring with 5 nodes and 3 rings with 6 nodes).

If only 3 rings are used, each may have 23/3=7 nodes (e.g., 1 ring with 7 nodes and 2 rings with 8 nodes).

5.5 Summary

In summary, we have presented three data flow designs. In A1, each node works on matches independently. A1 is fault-tolerant, and the FPGA nodes are well-utilized. However, A1 does not scale well to large numbers of templates because each node must store all the templates. A2 places all the nodes in a single communication ring. A2 scales well, but it is less fault tolerant than A1, and nodes may not always have work to do. A3, a hybrid of A1 and A2, consists of placing the nodes in numerous rings. A3 also scales well and is more robust to failures than A2 (but not as robust as A1). In A3, nodes could at time have to wait for work (as in A2).

A multi-stage mesh topology is also presented which illustrates the flexibility of the physical network used in the system.

6. Software Functionality

6.1 Introduction

The software provided by Myricom runs on a host machine and on the FPGA nodes. As shown in the figure below, the Myricom consists of four firmware components (labeled A, B, C and D). These four components run concurrently on the host processor, the host's LANai, the FPGA node's LANai and the node's FPGA, respectively.

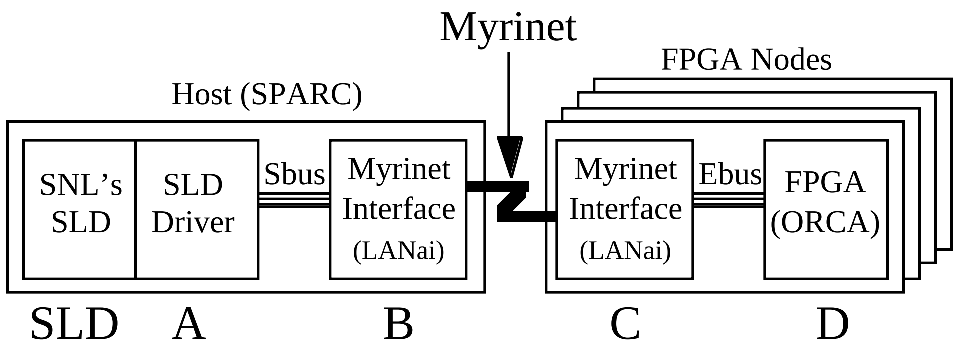

Figure 18: Software configuration (SLD by SNL; A,B,C,D by Myricom)

On the host machine, A resides on the Sparc and communicates with Sandia's SLD software. Component A also communicates over the SBus with B which resides on the host's LANai and is responsible for sending and receiving messages over the Myrinet link.

On the FPGA node, C runs on the LANai, and D runs on the ORCA FPGA.

6.2 Initialization

When an FPGA node is powered up, the ORCA initializes itself, using data from the EEPROM with D* that loads the LANai's SRAM with an initialization program, C*. This program has very limited functionality: it can respond to mapping messages, and it can handle boot messages from a host.

With C* installed, the network can be mapped, and the FPGA nodes can be identified. To map the network, A loads the host's LANai with B*, a mapping program.

6.3 Interfacing with the SLD code

After the hardware powerup of the FPGA nodes, any further processing done by the Myricom firmware is as a result of calls made by the Sandia SLD software to the following functions (which are all in A):

```
fpga_init_node_info();
fpga_load_template();
fpga_send_templates();
fpga_calc_sld();
```

When SLD calls fpga_init_node_info(), it sets off the following set of events. First, the LANai on the host is loaded (by A) with B, which begins execution by sending a boot message to each of the remote nodes. This boot message contains new software for the LANai on the node (C) and may also contain software to configure the FPGA (D). The LANai on the FPGA node (still using C*) replaces itself with C, which then loads the FPGA with D, and then waits for further messages. All FPGA nodes receive identical software (consisting of C and D).

Next, A uses B to send each FPGA node the source route for its “next” node (i.e., where it should send its answers). The choice of the next node depends on the design used (A1, A2, or A3).

After the FPGA nodes have been loaded, SLD calls fpga_load_template() once for each template that it wishes the nodes to remember.

For each template, the software on the host (A) stores the following data provided by SLD:

- bright mask	B(u,v) for u,v=0...31
- surround mask	S(u,v) for u,v=0...31
- template number	
- bias	Bias
- minimum threshold value	THmin
- maximum threshold value	THmax
- minimum bright sum value	BSmin
- minimum surround sum value	SSmin
- bit count of bright mask	BC
- bit count of surround mask	SC

In addition, component A calculates and stores several other values including:

- first non-zero row of either the bright or surround sum	
- bitmap that indicates which rows in the bright mask are all-zero	
- minimum shape sum value	SMmin
- maximum shape sum value	SMmax

Next, SLD calls fpga_send_template() which causes the templates to be sent one at a time to the software running on the nodes. (Depending on the data flow design being used, either all the templates will be sent to all the nodes (under A1), or some subset of the templates will be sent to each node (under A2 or A3)). Now the FPGA nodes are ready to receive match requests.

Finally, SLD will call fpga_calc_sld().When this function is called, SLD provides a pointer to the image, one or two ranges of template configurations to correlate the image with, and a pointer to where the answer should be placed.

As a result of the call to fpga_calc_sld(), a match task will be sent to a node. For A1, the match will be computed at that node and an answer sent back to the host; for rings of nodes as in A2 or A3, the match will continue on from the first node to all the other nodes in the ring and then back to the host.

Once the answer is returned to the host, component A places the best two matches in the location specified by SLD in the fpga_calc_sld() call, and returns.

Note that these software interfaces were designed to closely mirror the interfaces that the SLD code had with existing software.

6.4 Design details

From the perspective of an FPGA node, the difference between A1 and A2 or A3 is very slight. All the nodes are given return routes to use when handing off results (and the nodes don't know nor do they care if this route is to the host or to another node). In A1, each node will be given the complete set of templates whereas in A2 and A3 each node will receive a subset - but to the node this just changes the number of comparisons it must perform.

In A1, the data sent from the host to a node consists of the match task; the data sent from the node to the host contains only the two best matches (i.e., the answer).

In A2 and A3, the data sent from the host to the first node, the data sent between the nodes, and the data sent from the last node to the host are all identical in structure and consist of the match task and the two best answers so far. When the match task comes back to the host, it has passed through all the nodes and the two best answers so far will in fact be the two best matches of the image with all the specified templates. We could use this data structure for A1 as well (note that combining the incoming and outgoing messages for A1 results in the message used for A2 and A3), but in order to keep A1's messages as simple as possible, we chose not to.

In A2 and A3 as in A1 all the nodes are loaded with the same software, and for A2 or A3 the designation of "first" node and "last" node are done by the host, and is completely arbitrary and is also transparent to the nodes.

6.5 LANai Software

The code running on the LANai on the FPGA node, C, has numerous functions. First, it must handle messages received over the Myrinet. This processing is:

```
- if a message has been received
    - if it is a boot message (with new code to run)
        - reboot ourselves and run the new code (C and/or D)
    - if it is a mapping message
        - respond to the message with a mapping-reply message
    - if the message contains data
        - if it has a route to send data to
            - save the route for future use
        - if it has templates
            - store the templates
        - if it has a match task
            - save the image, and the data
            - place it on the task queue
```

Match tasks specify ranges of templates to use. When C receives a match task it places a separate entry for each specified template on a queue for D to process.

A queue entry corresponds to an image-template pair and consists of the following information:

- pointers to: image, bright and surround masks, threshold, and result buffers,
- bitmap that indicates which lines in the bright mask are all-zero,
- the template number (used to access other template information like the minimum bright sum and Bias),
- the image id,
- flags used by the LANai to know when a match task is complete and to aid in freeing buffers that are no longer in use.

Once a match task has been received and the image-template pairs have been placed on a data queue, the LANai's task is to ensure that this data is supplied to the FPGA at the proper time and with the proper pointers for processing to occur.

Ideally, the FPGA should have work to do at all times. Therefore, one of the most important jobs that the LANai on the FPGA node has is to keep the FPGA supplied with work. Recall that there are two tasks that the FPGA must do for each image-template pair. First, it calculates a shape sum (P1) and then it simultaneously calculates the bright and surround sums (P34). There are two additional computations that the LANai must perform. One is to calculate the thresholds from the shape sums (P2) and the other process, P5, is to calculate the two best matches and to report this to the host (or to the next node).

In addition to these computing processes (P2 and P5) the LANai also has a management process, P0, responsible for handling all incoming match tasks.

While the FPGA is working on task P1 or P34, the LANai on the FPGA node can be working on P2 or P5. The software on the LANai supports this by providing four queues for the four tasks (P1, P2, P34, P5).

The diagram below shows the flow of one image-template pair as it passes through the various processes and queues on the FPGA node.

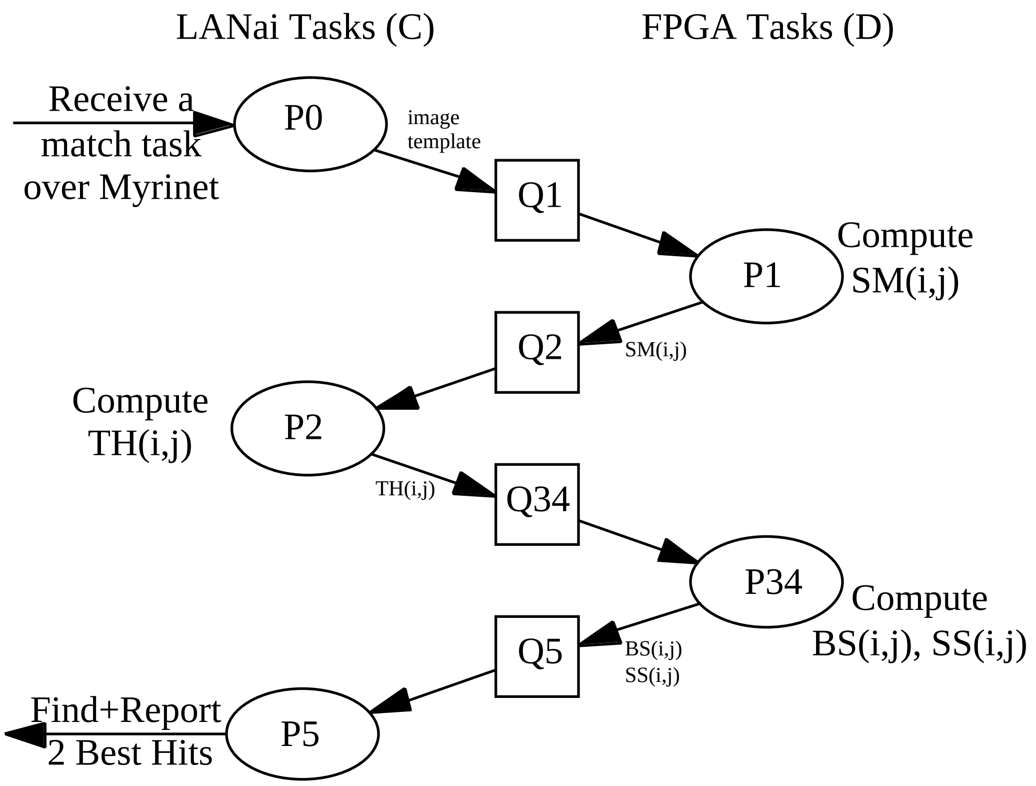

Figure 19: Data flow through the tasks and queues on the FPGA node.

Ideally, the FPGA should be kept well supplied with tasks, and any time that the FPGA is busy with a task (P1 or P34), the LANai should also be busy with a task (P2 or P5). The code on the LANai to support the processing shown in the figure has the form detailed below:

```
- if data on Q34 AND fpga_busy is FALSE
        - dequeue data off Q34
        - send data to fpga to perform P34
        - set fpga_busy to TRUE
        - set current_task to 34
- else if data on Q1 AND fpga_busy is FALSE
        - dequeue data off Q1
        - send data to fpga to perform P1
        - set fpga_busy to TRUE
        - set current_task to 1

- if data on Q5
        - dequeue data off Q5
        - analyze data for two best hits (P5)
        - if this match task is completed (i.e., last template of a set)
                - send answers to host or next node (P5)
- else if data on Q2
        - dequeue data off Q2
        - divide elements from fpga result (P2)
        - queue data on Q34

- if fpga_busy is TRUE
        - if FPGA is done (indicated by a set WAKE bit)
                - set fpga_busy to FALSE
                - if current_task is 1
                                - queue data on Q2
                - if current_task is 34
                                - queue data on Q5
```

By using this design with four queues, the image-template pairs flow easily through the four computational tasks.

The processes are independent and Q34 is checked before Q1, and likewise Q5 before Q2, so that we complete ongoing image-template matches before starting new ones.

Another interesting way to look at the data flow is to see what the LANai and FPGA are doing at any given time (see the table below). Because the FPGA can only perform one task at a time (either P1 or P34) and likewise the LANai can only perform one task at a time (either P2 or P5), any column in the table can have at most 2 entries - one indicating on which image-template pair the FPGA works and one on which pair the LANai works.

time slot:	0	1	2	3	4	5	6	7	8	9
process:										
FPGA P1	0	1			2	3			4	5
LANai P2		0	1			2	3			4
FPGA P34			0	1			2	3		
LANai P5				0	1			2	3	

Figure 20: task and image-template pair versus time slot.

At time slot 4, for example, the FPGA is performing task P1 for image-template pair #2, and the LANai is performing task P5 for pair #1.

The time to perform each of these processes may vary for each template and each template location due to the early-outs employed at each step of the correlation process. The time slots in the table above would be determined by the time taken by the slower of the two processes at work.

Ideally, we would like the FPGA to never have to wait for more data - that is, we would like the combined time to do P2 and P5 (the computations done by the LANai) to be less than the combined time taken by the FPGA to perform tasks P1 and P34 (meaning that the LANai would always be waiting for the FPGA with more data for it to work on).

Currently, the processing on the LANai is a bottleneck and if we could speed up the time taken for P2 or P5 (by moving some of the processing to the FPGA or the host machine), then we could speed up the total processing time. Recall that the process P2 is a division by a constant (BC) and subtraction of another constant (Bias) - a natural job for the FPGA.

7. Hardware

7.1 Structure

The hardware for implementing this system consists of FPGA nodes and baseboards.

The structure of the FPGA nodes is shown below.

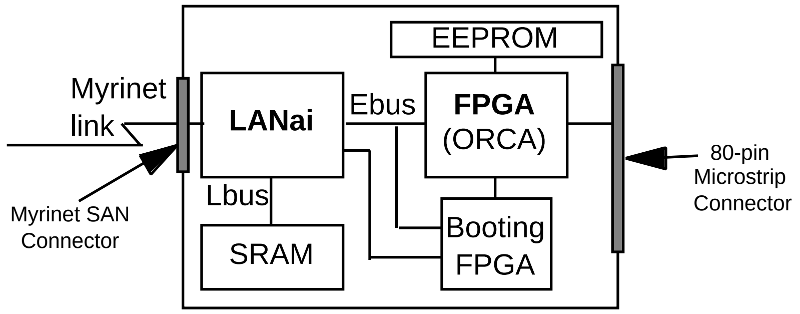

Figure 21: The structure of FPGA nodes

The baseboard, which supports 4 FPGA daughter boards, is a VME-6U board whose structure is shown below.

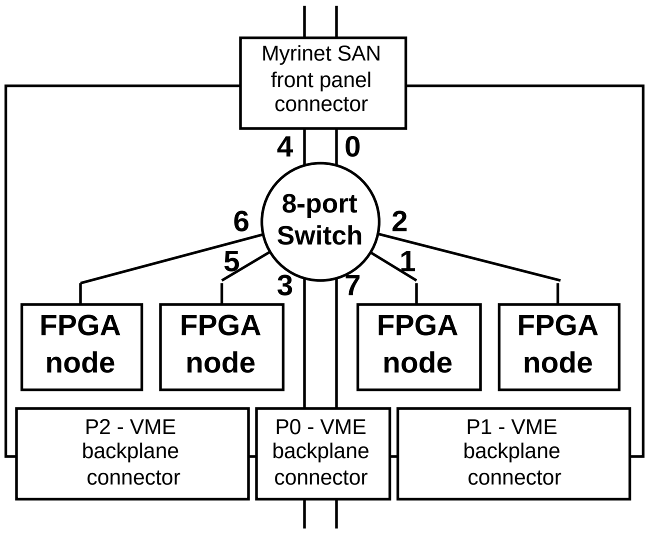
Myrinet SAN front panel connector
4
0
8-port Switch
6
2
5
1
3
7
FPGA node
FPGA node
FPGA node
FPGA node
P2 - VME backplane connector
P0 - VME backplane connector
P1 - VME backplane connector

Figure 22: The structure of the baseboard for 4 FPGA nodes

The following figure shows the actual hardware and the placement of its components. It shows a baseboard, with one (out of possible four) FPGA node plugged into it, and the other side of another node.

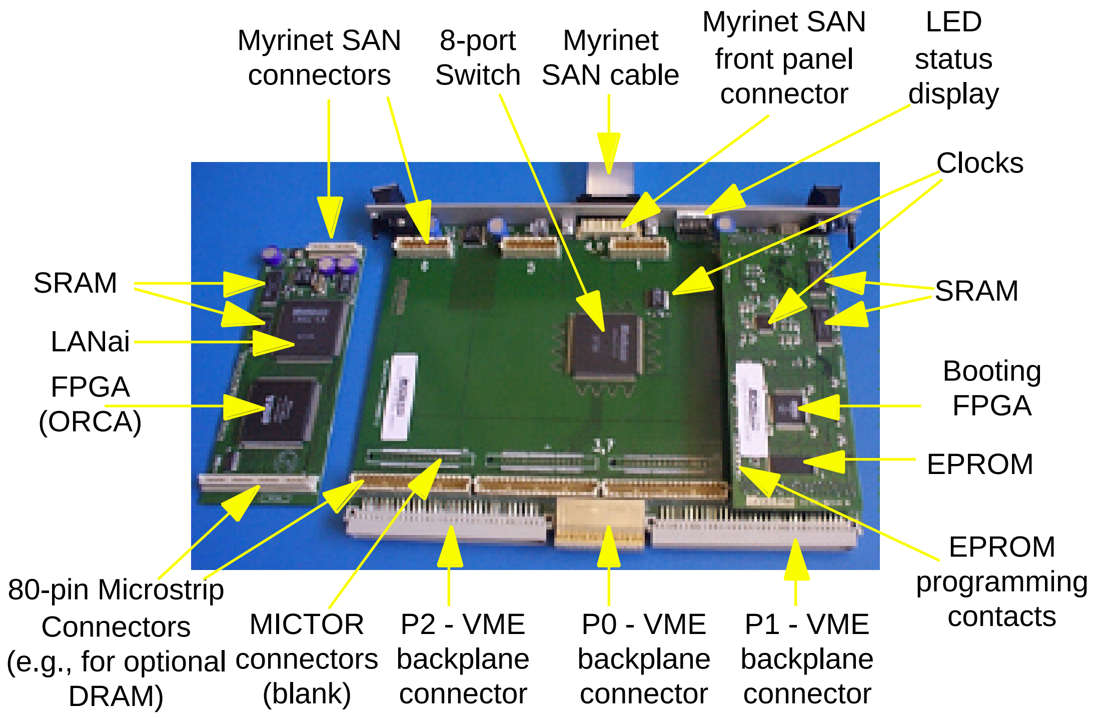
Myrinet SAN connectors
8-port Switch
Myrinet SAN cable
Myrinet SAN front panel connector
LED status display
Clocks
SRAM
LANai
FPGA (ORCA)
SRAM
Booting FPGA
EPROM
EPROM programming contacts
80-pin Microstrip Connectors (e.g., for optional DRAM)
MICTOR connectors (blank)
P2 - VME backplane connector
P0 - VME backplane connector
P1 - VME backplane connector

Figure 23: A picture of the baseboard with two FPGA nodes

This system is a two-level computing system whose first level provides the general purpose infrastructure of network interfacing, message handling, mapping, initialization, etc., by the LANai microprocessor, and second level provides the application-specific computing, which in this case is performed by the ORCA FPGA.

7.2 Initialization

Each of the FPGA nodes could be operated as a self-contained node. At powerup time both the computing FPGA (the ORCA) and the booting FPGA on each node initialize themselves using data from the EEPROM. Then, the ORCA uses data from the EEPROM to load the SRAM of the LANai with its initial program, the C* mentioned in the *initialization* section of the discussion of the *software functionality*.

Once this C* program is loaded, the LANai gets out of its RESET mode, and starts executing that program. From then on, the booting FPGA is waiting for commands from the LANai to reconfigure the computing FPGA, as needed (e.g., with D for ATR).

In our system, this LANai program, C*, depends on programs in the hosts to “program” the FPGA (by providing its configuration data, D) and to replace this C* by the runtime software, C. Loading the FPGA in this manner is done because this is a research and development project, and should be capable of supporting future evolutions as will be required by Sandia.

In a production system, the runtime software could be loaded directly from the EEPROM.

7.3 The Hardware

Each node has:

LANai4.3 RISC processor, 40 MHZ, 3.3V
Lucent ORCA FPGA with 40K gates, 3.3V, speed grade 3
SRAM, 512KB
Booting FPGA
EEPROM

The baseboard (M2M-VME-FB4) has:

Xbar8, 8-port switch
Micro-controller
LED

Four switch ports are connected to the four FPGA nodes (ports 1,2,5,6). Two other ports (ports 0 and 4) are connected to a SAN connector on the front panel, and the remaining two ports (ports 3 and 7) are connected to a P0 connector that is plugged into the VME backplane.

This arrangement allows dedicating a link to each node, accessing all nodes from any single link, or any combination in between. (In the ATR application the link bandwidth is low enough such that using one external link for all 4 nodes does not slow down the operation.)

Each daughter board is connected to the baseboard using two connectors, a Myrinet SAN connector (AMP's 40-signal microstrip connector) and an AMP 80-signal microstrip connector. Having a Myrinet SAN connector allows direct connection of the board to Myrinet SAN cables, should the need arise. The other connector links the ORCA with a slot on the baseboard for a Mictor connector that could be used for connecting additional daughter boards, such as for additional FPGA nodes or DRAM SIMMs.

The only use of the 80-signal connector on our nodes is to communicate between the node's ORCA and a microcontroller on the baseboard, which controls the LED display. This LED, visible on the front panel, is used to indicate the status of the system (such as to show which nodes are running).

The FPGA daughter boards also have contacts for (re)programming their EEPROMs.

A typical VME-6U 19″ subrack can house 21 boards. 16 of these may be M2M-VME-FB4 baseboards, connected to 5 M2M-VME-SW10 (or M2M-VME-SW12) as shown below. This arrangement yields 64 FPGA nodes in one 6U subrack.

At the conservative estimate of only 1,500 TSN, i.e., 1500 (template/sec)/node, this arrangement yields 96,000 (template/sec) per 6U-subrack that has only 16 baseboards.

By using the backplane links (over P0) with backplane (or bulkhead) switching, 21 baseboards with 84 FPGA nodes could fit into a single 6U subrack, yielding 126,000 (template/sec) per 6U subrack.

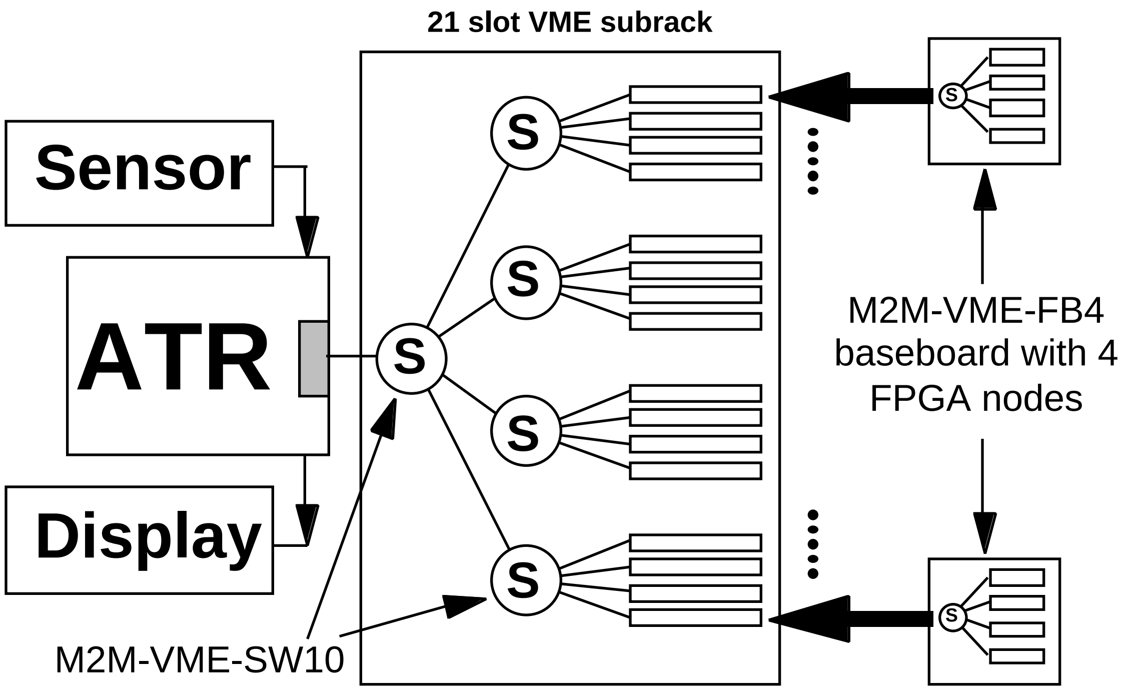

Figure 24: 64 FPGA nodes in a 21 slot VME-6U subrack, using front panel links

If so desired, we can change the form-factor of the FPGA nodes such that 8 of them would fit on a single baseboard that would be redesigned both for handling the new form factor of the nodes and their increased number, either by using two Xbar8 or a single Xbar16.

This would double the number of computing nodes per subrack to 128 or 168, depending on the use of front panel or backplane switches.

8. Performance (Present and Future)

8.1 Test Setup

The test and measurement setup is as shown in the following figure, where

Host Machine:	SparcStation IPX running SunOS 4.1.3
Host Interface:	Myricom M2F-SBus32B board with 512K memory
Switch:	Myricom M2FM-Switch8 switch
FPGA Node:	LANai4.1 running in 3.3-4V at 40Mhz
	ORCA FPGA 40K operating on 3.3V supply

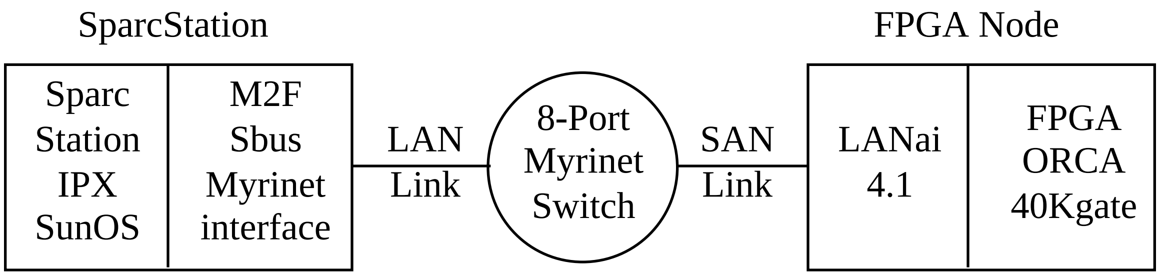

Figure 25: FPGA node test setup

The following measurements were made using the above hardware setup and customized software designed to accurately test the performance of the FPGA nodes, with minimal software overhead.

Notation: the performance of an FPGA node is measured by the number of template it can match in a second, written as TSN for "(templates/sec)/node".

8.2 Performance Without Optimizations (estimated and simulated)

Assuming no optimization, the number of cycles to perform each operation is given below.

P1 (shape sum): 11+8+(32*32+17)*21=	21,880 (cycles/template)/node
P34 (b/s sum) : 11+(8+8+32*32+17)*21 =	22,208 (cycles/template)/node
Total FPGA calculations	44,088 (cycles/template)/node

At 40MHz, a cycle takes 25nsec, hence 44088*25ns=1.102200 msec/template

1/(1.1022 msec/template) = 907 TSN

Without any further optimizations, this FPGA system can deliver ~907 TSN.

This figure was verified by using the ORCA simulator.

8.3 Skipping Zero-Mask-Row (estimated)

The ATR algorithm implemented in the FPGA nodes can take advantage of the sparseness of the templates. While computing P1 and P34 the FPGA algorithm will skip all-zero rows of the template.

The number of mask rows in 72 actual templates for each of 2 targets is 72*2*32=4608. Of these, there are 2206 non-zero bright-rows and 2815 non-zero surround-rows.

The non-zero bright-rows are 2206/4608=16/32 of the total.

The non-zero surround-rows are 2815/4608=20/32 of the total.

Using these ratios the cycle count is reduced to:

P1 (shape sum): 11+8+(32*16+17)*21 =	11128 (cycles/template)/node
P34 (b/s sum) : 11+(8+8+32*20+17)*21 =	<u>16832 (cycles/template)/node</u>
Total for the FPGA computations =	25272 (cycles/template)/node

At 40MHz 25272 cycles/template correspond to ~ 1583 TSN.

This is a 1.75X improvement over the non-optimized performance.

This performance can be achieved only if the FPGA is the pacing process (aka the "bottleneck"), which occurs if the LANai performs its computing tasks (P2 and P5) faster then the FPGA performs its tasks (P1 and P34).

8.4 Skipping Zero-Mask-Row (measured)

This skipping of zero mask rows was implemented in the FPGA firmware, and the performance was measured. 72 templates were matched with one image. This was repeated until 10000 matches were performed.

· When the division of P2 was done by the LANai while the FPGA was processing either P1 or P34, the performance measurements were less than the expected average.

  Average time/template: 0.9725 msec, corresponding to ~ 1028 TSN.

· Next, changes were made to the LANai software in the FPGA node to improve the performance by spreading the 21 divisions (for each search row) between P2 and P5, such that they required more balanced periods.

  Average time/template: 0.76016 msec, corresponding to ~ 1316 TSN.

- Since the division in the LANai is the bottleneck, it should be moved to the FPGA. The entire P2 task (the division and the subtraction of the Bias) should be performed by the FPGA, concurrently with other operations without slowing it. To estimate the resulting performance the following measurement was taken after turning off the division in the LANai (P2), simulating the performance as if the FPGA were performing the division.

  Average time/template: 0.6305 msec, corresponding to ~ 1586 TSN

  This matches the estimated performance of 1583 (templates/sec)/node.

By moving the entire P2 task to the FPGA, the performance should be higher than the measured 1586 TSN (which was achieved by a partial move of P2 to the FPGA).

8.5 Transposing Narrow Masks (future, estimated)

As outlined in Section 3 under Lesson 4, if a template is narrow, it is possible to transpose it and correlate it against a transposed image.

The following is the performance calculation using the same templates (transposed, off line, whenever needed for optimization).

```
P1  (shape sum): 11+8+(32*12+17)*21   =     8440 (cycles/template)/node
P34 (b/s sum) : 11+(8+8+32*18+17)*21 =   12800 (cycles/template)/node
Total FPGA calculation total =             21240 (cycles/template)/node
```

Transposing images may be performed by a straight char-char copy: 2704 read cycles and 2704 write cycles = 5408 cycles/image. The host can also send both the original and the transposed images.

Transposing the image has to be done only once for many sets of templates (72 templates for each target).

If the LANai on the FPGA node transposes the image, the cost to transpose should be divided by the number of templates that are to be checked. Therefore, the image overhead is probably on the order of 100 cycles per template.

At 40MHz 21340 cycles/template correspond to ~ 1874 TSN, a 1.14X improvement .

8.6 Skipping Invalid Threshold Lines (future, estimated)

If all the threshold values for an entire search-row are invalid (i.e., each is either below THmin or above THmax) then there is no need to perform P34 on that row since no hit found in that row can be valid.

Our data has:
(2*72 templates)*(16 images)*(21 (search-rows/image)/template)=48384 search-rows.
Of these rows, only 19543 have valid thresholds, or 40%.

With no optimizations we had:

P1 (shape sum): 11+8+(32*32+17)*21 =	21880 (cycles/template)/node
P34 (b/s sum) : 11+(8+8+32*32+17)*21 =	22208 (cycles/template)/node
Total FPGA calculation =	44088 (cycles/template)/node

At 40MHz, 44088 cycles/template correspond to ~ 907 TSN

If invalid-threshold search rows are skipped then we have:

P1 (shape sum): 11+8+(32*32+17)*21 =	21880 (cycles/template)/node
P34 (b/s sum) : 11+(8+8+32*32+17)*21*0.40 =	8880 (cycles/template)/node
Total FPGA calculation =	30760 (cycles/template)/node

At 40MHz, 30760 cycles/template correspond to ~ 1300 TSN, a 1.43X improvement.

8.7 Performance Summary

8.7.1 Present

No optimization (straightforward, brute force)	907 TSN
Estimated and verified by simulation.	
Skipping zero-rows:	
Measured with the initial P2 and P5	1028 TSN
Measured with the balanced P2 and P5	1316 TSN

8.7.2 Future

Estimated and measured without division by the LANai	1586 TSN
The division could be implemented by the FPGA	
Transposing masks+images	1.14X
Skipping invalid TH rows	1.43X
Total	2585 TSN

8.8 Future Improvement

The entire P2 task could be moved to the FPGA with no FPGA performance penalty.

Moving the entire P2 task to the FPGA will simplify the entire operation by having the FPGA perform (repeatedly) one task only, P1234, eliminating some of the queues, and significantly simplifying the LANai's coordination/management chores. This will leave the LANai with only the much simpler tasks P0 and P5, as shown below.

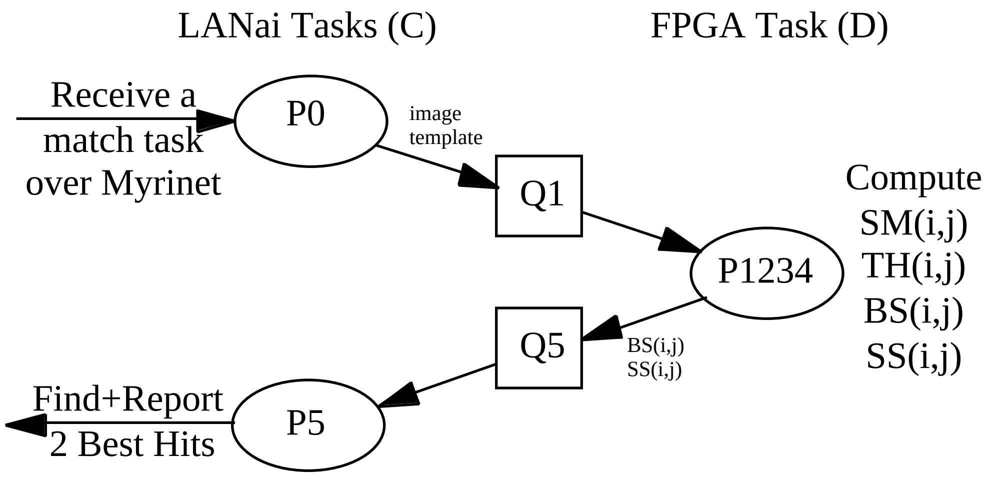

Figure 26: Simplified data flow through the tasks and queues on the FPGA node

If needed, the selection of the 2 best hits could also be moved from the LANai's P5 to the FPGA with no FPGA performance penalty. However, this may not scale easily to higher numbers of best hits.

As long as the FPGA can keep up with the LANai it pays to move tasks from the LANai to the FPGA.

9. Summary

The FPGA node consists of an FPGA (ORCA-40K gate) for computation, a Myrinet network interface (LANai4.3 and SRAM), and a small FPGA with some flash memory for booting. The computation FPGA has no memory directly attached to it; the computation FPGA must access the LANai's memory for template data and image data. The LANai has 512KB of memory attached which is used for network message data, program space, template storage, and image data. A single template requires 270 bytes, and there are 72 templates for each target configuration. The initial implementation supports 6 target configurations per node. Additional target configurations are supported by spreading out the templates amongst the processing nodes.

The correlations are mapped to a linear systolic pipeline. A high degree of parallelism is exploited. In addition to computing an entire row of correlation results in parallel (21 positions), the FPGA performs the address calculations, data loading, and correlations in parallel. Short inter-register paths allow the design to run at 40MHz, which is limited by the clock rate at which external memory can be fetched.

In sequential implementations there are three stages of the computation, thus there is plenty of opportunity for reconfiguration. The first stage consists of the accumulation of 8-bit data into a 16-bit accumulator, the second and third stages consist of comparing 8-bit values and conditionally incrementing counters. In the FPGA implementation, the second and third stage were optimized to be performed in parallel. Instead of reconfiguration for each stage, there is clever design of the processing element so that it can perform all these
3 stages, such that no reconfiguration is necessary, and hence no time is spent on reconfiguration instead of computation.

The compact processing element consists of one 8-bit data input register, one 8-bit accumulator, one 8-bit adder/subtracter, and two 8-bit counters. 21 of these processing elements along with the address generator reside on the FPGA while the divide operation is concurrently done in software in the LANai. There are 1024 (32x32) pixel locations in a mask, and 21 lines in the search area, and two processing steps and some overhead (~1K cycles), for a total of ~44000 (32*32*21*2+1000) cycles, which at 40MHz requires 1.1msec, yielding the rate of ~907 TSN. This was verified by simulations.

Improved performance has been achieved by several additional optimizations. These optimizations are similar to those that microprocessors use: exploiting the sparseness of the templates, and stopping the computation for a given test as soon as a processing stage fails ("early outs"). Since the correlations are in a linear pipeline we cannot exploit the sparseness in a given row, however it is easy to eliminate the calculation for a row if the template is entirely empty for that row.

Skipping zero rows of the template yielded the measured LANai-limited performance of 1316 TSN, which could be raised to 1586 TSN (measured) if some computing tasks were moved from the LANai to the FPGA.

To maximize this possibility, narrow templates could be transposed into short ones and rotated image data could be used for correlation. Another optimization is to abort the computation (an “early out”) when an entire row does not pass the shape sum test.

After these optimizations an FPGA node could process 2580 (template/sec)/node, if the data set that we received is indeed representative of the expected data, and if the LANai keeps up with the FPGA nodes.

Four FPGA nodes fit on a single VME-6U baseboard. Using 1316/2580 TSN as the present/future performance yields the performance of 5264/10320 template/sec by a 6U baseboard.

A standard VME-6U subrack has 21 6U boards, which may be all baseboards (with a total of 84 nodes) yielding the performance of 110500/216700 template/sec by the entire subrack.

If this performance is not sufficient, we can design another form-factor for the hardware and populate each baseboard with 8 FPGA nodes, doubling the performance for each baseboard and for the entire subrack.

Populating a subrack with 21 baseboards requires the use of switches outside the subrack. If it is preferred to use only switches that are internal to the subrack without using the P0 backplane, and using only A-links, then only 16 baseboards may be used in a subrack, with 64 FPGA nodes only, yielding the performance of 84200/165100 template/sec by the entire subrack.

Currently the bottleneck of the operation is the division which is still being done in software in the LANai. We could move the entire computing task P2 from the LANai to the FPGA, and would expect approximately 2500 TSN, yielding 210000 match/sec for a VME-6U subrack with 21 baseboards, each with 4 nodes.

The performance could be further increased if one were to reimplement the design with a larger FPGA, even without adding more external memory bandwidth. There is at least a factor of 21 of unexploited parallelism since rows are processed sequentially. Most of the memory fetches are common to the next row of computation, however by computing multiple rows at the same time, the chance that the shape-sum early out will occur decreases. Doubling the hardware on-chip might give close to 2X performance improvement, but 21X hardware increase will give only around 8X performance increase unless memory bandwidth is significantly increased, too.

10. Conclusions

In this report we have described a scalable FPGA system, not just a single accelerator node FPGA. This system is scalable by using an embeddable high-performance networking technology (Myrinet) that has programmable network interfaces. These network interfaces contain microprocessors for local control of the FPGA accelerators. This is an excellent example of two-level multicomputing where the second-level (the FPGA) can contain no control capability, and it is entirely dependent on the first-level (the programmable network interface).

We have demonstrated a scalable FPGA system for an automatic target recognition application. For that application we have measured 1316 TSN and can further improve it to reach 1536 TSN, and possibly further to 2580 TSN, yielding 165120 or 216720 template/sec by using a VME-6U subrack with 16 or 21 baseboards.

This performance scales ***linearly*** with the number of nodes. The only limit to its scalability is the ability of the host to dispatch and handle matching tasks.

We expect further planned design enhancements to increase the computational density advantage by another 33%. Even further performance advantages can be achieved by exploiting the remaining parallelism the design provides by using currently available, larger FPGA devices.

To achieve high performance there were eight main design decisions:

(1) use a proper system architecture (scalable);

(2) have a proper decomposition between hardware and software, not try to execute in FPGAs the complex but not computationally intensive parts of modules that are best left to software in host microprocessors;

(3) use dense packaging to achieve high performance for a given volume, microprocessors typically have a non-trivial amount of support chips;

(4) have short inter-register paths for fast clocking;

(5) have a design that takes advantage of high degree of parallelism;

(6) have a design that can adapt to the dynamic variances in the data to eliminate excess computation, just like in microprocessor software;

(7) do not neglect start and finish overheads; and

(8) minimize reconfiguration since most current FPGA devices reconfigure slowly.

[1v1]