Pergamon

Computing Systems in Engineering, Vol. 6, Nos 4/5, pp. 401–408, 1995
Copyright © 1995 Elsevier Science Ltd
Printed in Great Britain. All rights reserved
0956-0521/95 $9.50 + 0.00

0956-0521(95)00029-1

A HETEROGENEOUS COMPUTER VISION ARCHITECTURE: IMPLEMENTATION ISSUES

Henrique Dinis Santos, José Carlos Ramalho, João Miguel Fernandes and Alberto José Proença

Department Informática, Universidade do Minho, 4719 Braga Codex, Portugal

Abstract—The prototype of a heterogeneous architecture is currently being built. The architecture is aimed at video-rate computing and is based on a message passing MIMD topology at the top level—transputer based—and on VLSI associative processor arrays (APA, SIMD structure) for low level image processing tasks. The APA structure is implemented through a set of 4 VLSI chips (GLiTCH) containing 64 1-bit processing elements each. This communication addresses some issues concerning the implementation of the first prototype, namely those related to:

—the design and integration of the APA controller unit, which provides the required interface between the APA, the MIMD topology and the video image interface;

—the evaluation of the GLiTCH chip through an emulator based on transputers and fast programmable devices; the emulator was designed to be flexible enough to evaluate later modifications to the GLiTCH design;

—the design of an integrated set of software development tools containing a structured editor—syntax oriented, with a visual interface/programming interface—and a cross compiler and debugger.

1. INTRODUCTION

Video-rate computer vision is a highly demanding computing field, requiring high performance heterogeneous architectures and new software and programming paradigms. The prototype of a research architecture based on a message passing MIMD topology at the top level—transputer based—and a SIMD structure for low level image processing tasks—based on an associative processor array (APA) implemented with a set of 4 VLSI chips—is currently being built.

The overall project started in 1987 with the design of the APA chip—GLiTCH—and its functional evaluation through a simulator developed in C.[1,2] Each GLiTCH chip contains 64 1-bit processing elements (PE) with a 1-D direct connectivity between PEs, a search data aligner and a video shift register 64×8 bits to input/output the data (video images). Each PE has 68 ternary digits of associative memory on-chip plus some RAM off-chip and a 1-bit ALU with 1-bit registers. A comparison of image processing architectures based on massively parallel structures using the SIMD paradigm shows the relative merits of the GLiTCH architecture in an image processing environment.[3] Performance estimates based on the first set of DARPA Image Understanding Benchmarks also show that a GLiTCH based architecture performs well on most of the benchmark tasks.[4]

To integrate the APA based on GLiTCH and the MIMD topologies, a controller unit was designed and implemented at Universidade do Minho (UM),[5] to provide that interface and to issue the required control signals. The APA structure requiring control signals includes not only the GLiTCH chips and the associated external RAM, but also a data routing network to provide fast inter-PE communication links, a data store to hold the patterns and to communicate between the APA and the MIMD topology, a data shifting unit and the video bus interface. This team project across national boundaries raised implementation issues presented later in this communication.

The VLSI GLiTCH chips are being manufactured and tested under Eurochip. To avoid timing dependencies on this product, it was also decided at UM to build a GLiTCH emulator—based on transputers and fast programmable devices, PALs and FPGAs—flexible enough to evaluate later modifications to the GLiTCH design. The overall structure of this reconfigurable emulator and some of the trade-offs of a custom design vs a general purpose one are analysed.

When the controller unit was built it was necessary to test it together with the APA structure. Testing of most of this complex hardware structure was performed by the available transputer in the board, through microcode downloading and single step microinstruction execution. To test the overall prototype it was decided to create a software development environment that would allow execution of most of the programs and routines that have been previously developed for the simulator (the most relevant ones are described in Refs 6 and 7). The

email: aproenca@ci.uminho.pt

GLiTCH instruction set and the programming environment provided by the latest version of the GLiTCH simulator[8] was the starting point to design a new integrated set of development tools, containing a structured editor—syntax oriented, with a visual interface/programming interface—and a cross compiler and debugger. The compiler also generates microcode for the controller unit, coping with the superscalar and superpipelining features of the underlying architecture.[9] Its current state closes this communication.

2. OVERVIEW OF AN INTEGRATED GLITCH BASED SYSTEM

The dual bus Heterogeneous computer Vision Architecture (HVA) that has been proposed[10] includes dedicated chips for digital signal processing, APA chips (GLiTCH) for low-level image processing, and a transputer network for higher level tasks; a parallel bus is used for video images' communication, while a serial bus transmits control and synchronization signals in the transputer network. A proper interface between the SIMD and MIMD topologies was designed and included in the controller of the APA module,[11] to provide an integrated vision environment supporting diverse programming approaches.

The HVA based on the APA chips heavily depends on pipeline processing, from image loading to high-level vision processing. Images are input to the APA through an 8-bit wide video shift register (VSR), where the shifting in is done concurrently with the shifting out and with computation on the Processing Elements (PE), enabling an image to be processed, while the next image is loaded into the APA and the previous image is being unloaded from the APA.

The APA unit contains an array of 1-bit PEs—as many as possible to fit in a single chip—each one with its local CAM. The CAM size—64 digits for the data CAM and 4 digits for the subset CAM—was chosen to allow word-parallel matching of an external pattern to the PEs local memory. Since the supplied patterns use ternary logic—"0", "1" and "x" (don't care)—the data CAM requires write/match patterns of 64 ternary digits (128 bits), and the subset CAM a further 4 ternary digits (8 bits). To reduce the data bandwidth, CAM array patterns longer than 16 ternary digits (32 bits) are not allowed, and a search data aligner—the pattern broadcast logic unit, PBL—was added, whose main function is to route the 32 pattern signals at the chip pins among the 128 CAM data bits. To hold partial results, the inclusion of some RAM per PE is also included off-chip, while still retaining the advantages of CAM for the data being processed.

Processor arrays for image processing are often mesh connected due to the two-dimensional nature of images. However complex interconnection schemes become difficult to implement in a VLSI chip due to the limited number of available pins, and they are not always the optimum connectivity. A 1D connectivity scheme may be more suitable to implement in a VLSI APA, and its disadvantage can be offset if fast barrel shift mechanisms are provided, either built into the CAM array of the APA chip, or through an external data routing network. A programmable data routing network can provide a fast 32 bit interchip data path between the PBL of each APA and its neighbour, and also to a 32-bit data routing bus.

A data store unit connected to the data routing bus holds the 16 ternary digits required by the APA, and interfaces the APA module with the MIMD structure. Some additional logic is also included to hold address values and to perform address increment/decrement operations, to implement parameter passing to subroutines at the SIMD programming level.

Scalar processing is provided by a host transputer with floating point capabilities, which can directly transfer data to and from the data store. For simpler operations a scalar unit to shift and test scalar values is also included in the APA module.

A controller unit is used to generate all the required control signals to select the operations of the functional units described above and to supply the addresses and values of operands, in an appropriate sequence. This unit also provides the interface between the APA module and the transputer network.

Figure 1 gives an overview of the main functional units in the APA module. The video frame buffers with ROI (Region-Of-Interest) capability—implemented separately and controlled by another transputer—provide the video image interface through a dedicated video bus.[12]

3. THE APA CONTROLLER UNIT

A complex controller unit was designed and implemented to control the functional units described above, generating the necessary control signals in an appropriate sequence, and with a performance aimed at a 50 ns cycle time. The initial APA controller specifications were taken from the simulator available at the time,[13] and some features were added: hardware support to implement nested subroutines with parameter passing at the SIMD programming level, additional high level control operations such as CASE and DO WHILE, and provision for an external scalar processor.

In a conventional microprogrammable controller approach, a simple sequencer—a microprogram counter, an incrementer and a multiplexer—generates the addresses to a micromemory, where a program with the required microinstructions is stored. This microprogram is responsible for fetching the "processor instructions", decoding and executing them, in a continuous loop. However, the time penalty for fetching and decoding each instruction is too high, since the GLiTCH array expects a set of control

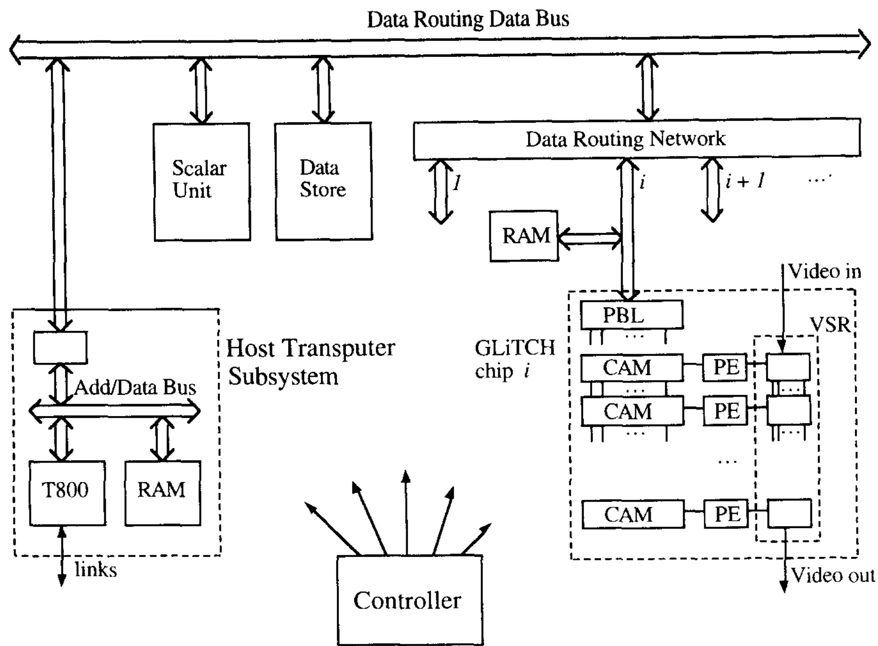

Fig. 1. Overview of the APA module.

signals every 50 ns; to add instruction caching and extra circuitry to deal with faults would mean a more complex overall circuit with longer processing latency. The instruction fetching and decoding were removed from the microprogram in the GLiTCH controller, requiring the compiler to directly generate a separate microprogram for each application. However, the real-time vision computing environment is a very special case, quite suitable for microcode compilation: in most situations the program needs to process each image in a very short time (ms) and then simply repeat the whole process. At 25 Hz frame rate, the same algorithm is applied for every image at every 40 ms; at 50 ns cycle time, this corresponds to the execution of up to 800,000 microinstructions per frame. Current experience running the GLiTCH simulator shows that a 32 kword micromemory is enough for all the applications that were developed so far. Whenever longer algorithms are required, a fast link with the host transputer may be required to allow the host to load new programs in the micromemory. To further reduce the micromemory requirements a complex micromemory sequencer is used to directly implement HLL control structures, including loop and nested subroutine support. This device can reduce quite considerably most of the unnecessary microinstruction duplications.

From the informal and bottom-up specification of the controller, a Harvard VLIW (Very Long Instruction Word) architecture was implemented, with the block diagram shown in Fig. 2. Note the decentralized nature of the controller unit, since the micromemory issues instructions to be further decoded, rather than issuing direct control signals. This decentralization of the control signal generation reduces by almost 50% the overall micromemory, also reducing the pin count on the custom and semi-custom chips. The micromemory width was also reduced from 128 bits to 96 bits by sharing some microinstructions fields between different units, based on the respective usage frequency and exclusion properties. The validation of those assumptions requires the feedback of using the system in real applications.

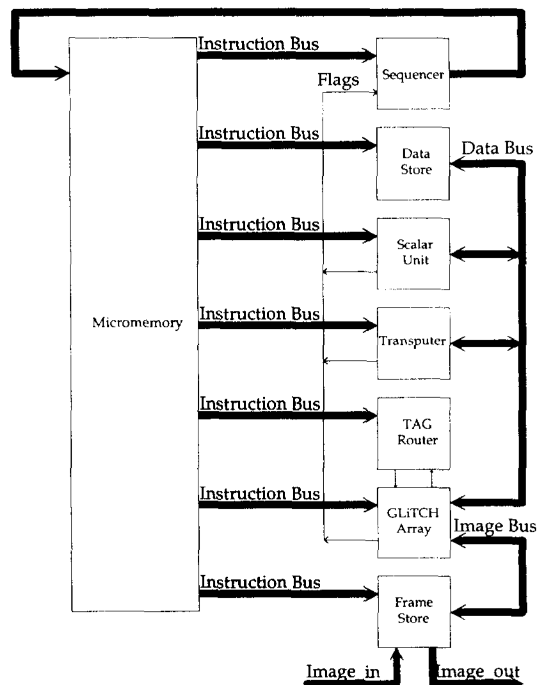

Fig. 2. The controller block diagram.

The APA functional units interact with each other to execute most instructions. A MATCH instruction, for instance, requires access to the data store to get the required pattern to load into a certain CAM position in the GLiTCH chips, through the data routing network. This simple operation may require some clock cycles to be accomplished, but it can be all specified in a single microinstruction. To keep the microinstruction stream at a rate of 1/50 ns, the APA architecture requires several pipeline stages; each functional unit needs its own control information to be set at different clock cycles, with extra pipeline registers being introduced to achieve the "one operation one microinstruction" paradigm.

Adding pipeline levels increases the throughput quite considerable, but it also has some disadvantages, mainly the branch dependencies on conditional operations. To reduce this penalty, some conditional operations are performed directly at the GLiTCH chip level without the controller intervention, by internally feeding the tag results. To implement other high level control structures the hardware on the first prototype gives no support to check the pipeline consistency, leaving to the compiler that task when implementing delayed branches.

Communication between the low level processing structure (SIMD) and the higher level (transputer based) is achieved in two ways:

- control and synchronization signals flow between the transputer and the APA controller, either to request the attention of the transputer, or for the transputer to request access to the APA data routing data bus (these signals are shared with the transputer activities in the role of initializing, testing and monitoring the APA module working);
- program and data interchange; whenever some high level vision task requires a low level activity, the transputer downloads the associated microprogram (or sets the sequencer microprogram pointer if the microcode is already loaded), and transfers the data patterns into the APA data store; the results back from the APA processing can be read by the transputer by accessing the same data store.

One of the main implementation issues with the APA module refers to its testing during the assembling phase, since it uses several complex VLSI devices from different manufacturers, including those custom designed for this project (the GLiTCH chips). This task is simplified when the same specification environment is used and when the design team works very closely. Neither of these characteristics apply to the APA module design, and no hardware simulation was performed for the whole APA module (only for those parts implemented in programmable logic devices). The GLiTCH chip design started back in 1987 at University of Bristol (UB), UK, and the latest version was produced at UMIST, UK, using different specification tools; half the APA module—the GLiTCH chips and the associated RAM, the data routing network, the clock generator and the video interface—was designed at UB, UK, from 1990 (the data routing network) until 1992, with no formal specification (only textual descriptions, logic diagrams and programming tables for the programmable logic devices); the same applies to the other half of the APA module, designed at University of Minho (UM) in 1991/92. Assembling and testing the first prototype suffered delays due to some less clear design details, as a result of this lack of a consistent specification notation across the design teams. During the implementation of the APA module, a formal description of the module started to be produced, in Verilog at UB, and in VHDL at UM.

Formal hardware description languages offer much more consistency during the design cycle, allowing the use of the same description language for the simulation, emulation and synthesis. They also have a shorter design time, better documentation and tools for partial automatic synthesis. The development of a behavioural model for the whole APA module is almost finished at UM, in VHDL. This model puts together different modules, with different characteristics; some modules are more adequately synthesized by software, as a scalar computer program, others fit better on ASICs components, while the rest can be implemented using off-the-shelf components. This system approach will be relevant to identify the weak points of the first prototype and to follow a top-down approach in the design and evaluation of the next GLiTCH system generation. This methodology also promotes investigation into different synthesis strategies, as well as the impact some specification changes have on the hardware.

From the begin of the design cycle the system testability after assembling was considered an important goal. As a standard design style, the use of Boundary Scan Test (BST) techniques was first considered. However, most of the APA module components (the transputer, the VLSI sequencer, the GLiTCH chips and the standard logic devices) do not support such test methodology, making it useless for this prototype design. The current prototype supports a test mode, in which the transputer can access all the system units. A set of programs was developed to test and to diagnose the micromemory and each functional unit, by making them execute individually a piece of microcode under the transputer supervision. This procedure uses the same facilities that are used to load an APA program, plus the insertion of microprogram breakpoints.

4. THE GLITCH CHIP EMULATOR

The first prototype of this heterogeneous architecture uses 256 PEs, implemented in 4 VLSI GLiTCH

chips. While the chips are being manufactured (under Eurochip) and tested (at UMIST and UB) it was decided to build a GLiTCH chip emulator to test the overall APA module, including the controller functionality and the MIMD interface.

The specification of the emulator included some specific guidelines; namely it should:

- have a general purpose structure to be re-used later to emulate other digital systems;
- be reprogrammable to evaluate later modifications and improvements of the GLiTCH chip;
- be cost effective to afford a separate emulator per GLiTCH chip.

While the first two guidelines suggested the use of complex programmable logic devices (CPLDs), the last constraint suggested the use of a mixed-mode approach: to implement the time critical parts using the CPLDs, while the less critical would be implemented by software running on a microprocessor. An emulator built only with CPLDs would require a large number of these devices to emulate a complex digital system; complex PAL devices are very powerful with combinatorial logic, but they have very few registers (usually under 100); on the other hand FPGAs have a large number of simple cells with registers (over 1000), but strong limitations in internal connectivity. Adding a microprocessor to the emulator architecture increases flexibility and allows the emulation of any medium to highly complex digital system, provided the overall system to be emulated can run on slower clock speeds. To minimize this potential drawback, an appropriate choice of the microprocessor might allow a speed-up improvement through parallel processing in a multiprocessor environment.

Following these guidelines, the design of the emulator architecture was based on one transputer and links to connect to external transputer systems, and on programmable devices—a variable number of complex PAL devices (such as the MACH from AMD) and FPGAs (such as the LCAs from Xilinx). Two main issues have not been addressed yet:

- how to cope with a slower clock speed;
- how to program the emulator.

The maximum clock speed of a standard emulator can be predicted if it relies only on hardware devices; however, when the emulation is performed with software parts, the clock speed is related to the time required to execute the slowest function, which can be undesirable, particularly if that function occurs very seldom. Alternatively, and if the device being emulated allows a clock with a variable period length, the emulator could generate this irregular clock, by using the transputer to set/reset a bit on a particular output port.

In the GLiTCH emulation project the time critical parts were identified as being the video interface: the internal video shift register in GLiTCH is shifted at the video clock rate specified by the external video source, which makes it impractical to slow down this clock frequency. This block of the chip is implemented using a FPGA device. Most of the remaining functional blocks were emulated through software executing in the transputer,[14] and through irregular block pulse generation, as described before. Since the vision prototype uses 4 GLiTCH chips, that are supposed to run synchronously, the pulse clock generation is also synchronized among the 4 emulators according to the slowest one.

The hardware structure of the emulator is now defined and clear. It supports emulation of a large number of medium to high complex digital systems, being the GLiTCH chip one of those examples. To program this system for a specific emulation task there are two main approaches:

- structured top-down, using the same language environment to program all devices (including the microprocessor);
- bottom-up, partitioning a priori the emulation tasks for the separate devices, and programming each of them using their specific programming environment.

The structured top-down is the more flexible one, but it adds some extra complexities. The choice of the programming environment also plays an important role: either a programming environment typical for a microprocessor (Pascal, C, or even Occam[15]), or an environment more suitable to interface with PLDs programmers. The first alternative requires conversion tools from a computer programming language to a hardware device programming language, and the optimization of the gate usage in PLDs is quite complex, due to the abstraction level of a computer programming language. A much more attractive approach is to use a standard hardware description language, such as a subset of VHDL with automatic tools to generate the programs for the CPLDs, to program and evaluate the most suitable mapping of the emulation tasks into the several devices, including the microprocessor. This approach requires compiling the VHDL language into the target microprocessor, or pre-compiling VHDL into a HLL program (a pre-compiler into C is being built at UM). An additional feature of this approach is that a single specification program is made in VHDL and then the simulation, emulation and implementation—using one out of several circuit integration alternatives—can be performed without having to re-write any line of code.

The first prototype of the GLiTCH emulator followed the bottom-up approach, since the required tools for the VHDL conversion were not available yet. Once these tools are built the emulator will be programmed using this novel approach.

5. THE GLITCH DEVELOPMENT ENVIRONMENT

The first implementation of the GLiTCH development environment (GDE) is basically a user friendly programming environment. It provides the user a set of tools to help to develop APA progrms in the assembly level language defined by the simulator, producing machine code that should load and run on the available hardware. The use of the language defined by the simulator was controversial (the simulator does not map the real hardware), creating difficulties when translating simulator instructions into the available hardware without performance degradation. However, the decision was made to allow reuse of most of the programs and libraries produced in the past to run on the GLiTCH simulator. A comparative study between the simulator and the actual implementation identified two types of problems:

- conceptual problems: each APA unit is treated separately in the simulator, and the superscalar and pipeline features are not included; instructions dealing with the memory require their syntax to be altered;
- usage problems: since all simulator programs are compiled with a C compiler, GLiTCH programmers prefer to use the C syntax to specify the control flow rather than the available and similar control instructions of the assembly level; all these programs will not run on the first prototype.

A new release of the simulator if required which includes the main modifications and yet would keep compatibility with most existing programs. This proposal came from the development of GDE, where the syntax of some instructions is modified, some instructions are merged (making some hardware features transparent to the user), and some other are eliminated.

This integrated environment in GDE contains two main modules:

- the environment's front end, which includes a structured syntax directed editor; the editor's embedded syntax was extracted from the context free grammar of the GLiTCH-based APA assembly language (GAL); this editor provides immediate syntax checking and automatic semantic error checking and warning;
- the environment's back end, which includes a cross compiler for the GAL, specified through a set of attributes' equations; while the actual version only generates microassembly code for the debugger,[16] next version will also include the generation of executable binary microcode and will cope with the transputer interface.

The tool used to implement GDE was the Synthesizer Generator (SGEN).[17,18] SGEN is specially oriented towards the development of structured editors, through a syntax directed approach (the syntax of a particular language, in this case GAL). To create an editor with SGEN its different parts must be specified using the specification language supplied with the tool, which is a very high level and functional language (the higher the language level, the easier the project management).

GDE is a structured editor generated by SGEN to perform the lexical, the syntactic and the semantic analysis at run-time and, simultaneously, to generate code as a side effect, following the paradigm of incremental compilation. This paradigm allows immediate feedback to the user when introducing the source code, by presenting the object code in a different window and issuing warnings whenever a mistake is made.

When writing a program for the APA module in the GDE environment, the user has two alternatives to introduce it through the editor:

- introduce the text (from a file or from the keyboard) that will be parsed according to the specification of the language context free grammar;
- write the program through the use of the supplied templates and mouse clicks, using a visual programming paradigm.

From this concrete syntax, GDE produces an internal representation, the abstract syntax. Acting on it, GDE can generate views to present to the user through windows. Current version shows the user two windows: the first on the left is the source window, where the user can edit a program in source language (the typing can be limited to the supply of arguments, since the programmer can use templates); the second window, on the right, is the object code window, showing the generated microcode in mnemonics (a microassembly language) while the user types a program in the first window; in this window only selections are allowed (code generation is entirely controlled by GDE). Although no interaction is allowed in the object code window, every item (item stands for any structural element of the language) in this window is related to another and correspondent item in the other window. Thus, the selection of an item in the object code window provokes the corresponding selection of an item in the other window (Fig. 3).

With this limited interaction syntax errors are discarded, and only some semantic errors (related to the supplied arguments) can occur. When they occur GDE types warning messages into the source code window, positioned at the statement where the error occurred. These messages will stay displayed until the errors are corrected.

```
GliTch PROG1;
DEFINITIONS
  pos1 : 4;
  pos2 = 10101100;
  pos3 = 11110000
BEGIN
END.
```

```
; Program Header
        NAME PROG1
        CORG  20
        DORG  10
        START 0
; Data Definitions
pos1    ds   4
pos2    dw   10101100
pos3    dw   11110000
; GliTch Instructions
LABEL   SEQ   addr   (CC) DS   addr   SU/TR  GLITCH
```

Positioned at contstmt

Fig. 3. The GDE user interface.

6. CONCLUDING REMARKS

The first prototype of the APA module to interface to a transputer network in a video-rate computer vision system is in its final phase of assembling and testing. With 256 PEs in 4 VLSI chips and additional blocks, the more relevant implementation issues were presented in this communication. As any other 1st generation machine, this will provide enough data to prepare a much better 2nd generation, in terms of functionality, speed, simplicity and easy of use.

The re-writing of the whole APA module specification using a single language environment, VHDL, is an essential step towards a faster and more reliable implementation of the 2nd generation. Together with an invaluable evalation tool (the emulator), which will be automatically programmed from the specification tool, the assessment of alternative configurations will be performed with more precise results (than using a simulator), and closer to the real implementation.

The existing software development environment (GDE) has proved to help programming a SIMD structure through increased productivity. One of the more relevant features of the GDE is its powerful interface, providing visual programming facilities together with an incremental compiler that allows on-line error checking and warning. The GDE is not complete yet (it does not address all the APA modules on the board designed in Bristol), and it requires further refinement in terms of pipeline optimisation. The following steps will include the porting of an architecture independent vision language, such as Apply,[19] and the definition of a higher level user interface for parallel computer vision programming.

Acknowledgements—The authors wish to acknowledge the financial support of the Junta Nacional de Investigação Científica Nacional (JNICT), Portugal, for this work.

REFERENCES

1. A. Duller, A. Morgan and R. Storer, "Associative processor arrays: simulation and performance estimates for image processing", Proceedings of Alvey Vision Conf. 87, September, pp. 139–145.
2. A. Duller, R. Storer, A. Thomson, E. Dagless, M. Pout, A. Marriott and J. Goldfinch, "Design of an associative processor array", *Proc. IEE, E*, **136** (5) (1989).
3. A. Duller, R. Storer, A. Thomson and E. Dagless, "An associative processor array for image processing", *Image and Vision Computing*, **7** (2), 151–158 (1989).
4. M. Pout and A. Duller, "Benchmarking an associative processor array for vision", *Applied Mathematics and Computer Science*, **3** (1), 141–154 (1993).
5. A. Proença, H. Santos and E. Dagless, "Microcontroller strategies in an associative array processor for computer vision", Proceedings of Barnaimage '91, ESPRIT-BRA Workshop on Specialized Processors for Real-time Image Analysis, Barcelona, España, September 1991.
6. R. Storer, A content addressable parallel processor and its application to synthetic image generation, Ph.D. Thesis, University of Bristol, 1991.
7. M. Pout, Performance evaluation of an associative processor array for computer vision tasks, Ph.D. Thesis, University of Bristol, 1992.

8. A. Duller, R. Storer and M. C. Ribeiro, "The improved GLiTCH simulator. V.2.2", University of Bristol, August 1992.
9. J. Ramalho, A compiler for GLiTCH, (in Portuguese), M.Sc. Thesis, Universidade do Minho, 1993.
10. R. Storer, M. Pout, A. Thomson, E. Dagless, A. Duller, A. Marriot and P. Hicks, "An associative processing module for a Heterogeneous Vision Architecture", *IEEE Micro*, **12** (3) 31–41 (1992).
11. H. Santos and A. Proença, "GLiTCH System Controller: General Description—Version 1.00.00", Internal Report DI-EC02/91, Dep. Informática, Universidade do Minho, April 1991.
12. Datacube Inc, "MAXbus specification", Tech. Report SP00-4, December 1987.
13. A. Duller, "GLiTCH Simulation—Version 2.1", University of Bristol, January 1989.
14. M. Alves, Emulação de um processador associativo vectorial—GLiTCH, Rel. Estágio de Lic., Universidade do Minho, Portugal, 1992.
15. I. Page and W. Luk, "Compiling occam into FPGAs", in *FPGAs*, Abingdon EE&CS Books, 1991.
16. P. Hanacek and H. Santos, "GLiTCH system controller: GLiTCH debugger and loader", Internal Report RI-EC01/93, Universidade do Minho, 1993.
17. T. Reps and T. Teitelbaum, "The Synthesizer Generator: A system for constructing language-based editors", *Texts and Monographs in Computer Science*, Springer-Verlag, 1989.
18. T. Reps and T. Teitelbaum, "The synthesizer generator reference manual", *Texts and Monographs in Computer Science*, Springer-Verlag, 1989.
19. L. Hamey, J. Webb, I.-C. Wu, "An architecture independent programming language for low-level vision", *Computer Vision, Graphics, and Image Processing*, **48,** 246–264 (1989).