Visual Debugging for a Pyramidal Machine

A. Biancardi and M. Mosconi

Dipartimento di Informatica e Sistemistica University of Pavia

Abstract

This paper proposes a novel approach to program development for highly parallel architectures, primarily as far as debugging is concerned. The visual nature of the debugging stage, when dealing with image-processing algorithms, is heavily supported so that all the relevant information, which is generally either hidden or presented without its logical structure, is made available to programmers.

1: Introduction

Massive parallelism is a well established approach to get good performance in low and intermediate image processing and in many other computationally intensive problems. The design of a new highly parallel architecture with some degree of complexity (like a real or a simulated pyramid) and in particular the study of an efficient utilization of such a system require an adequate development environment for the generation and the testing of programs.

The traditional development instruments (mainly loosely integrated tools with textual interfaces for supporting the various phases of software production) do not sufficiently meet the needs of programmers because of their limited communication power: first of all a graphical depiction of data (instead of a textual one) is mandatory when dealing with image computations; secondly, facilities must be provided to represent and retrieve information at the required level of abstraction.

The availability of easy to use, portable and efficient tools is clearly important to reduce the cost of algorithms and programs development (which is an increasing fraction of the total Machine Vision Application cost); so a considerable number of new application development systems [1,2] has been produced, all sharing the same productivity goals and the common exploitation of the so called visual technologies [3].

In this work a novel approach in designing software environments for massively parallel fine grained machines is proposed. We present the modular and portable software system built, in Pavia University, for the under-development PAPIA2 machine, devised to provide an easy and effective framework to generate and debugging (testing) programs while graphically monitoring their executions in a natural and profitable way.

The key units in the resulting debugging process are therefore images, representing the status of various registers and memory locations of the processors in the machine (organized in square arrays). The user expects the system to show a predicted behaviour and monitors program executions at run-time: they are just (even small) variations of colours or shapes (with respect to the expected image) which are easily detected by his eyes and can reveal a bug and its nature.

As a matter of fact this one is not an intelligent debugger with respect to fault localization or remediation: instead it is a visibility tool[4], capable of making available (and understandable) information that is usually lost or hidden from programmers. A major skill consists just in representing data the way they are logically organized (binary or grey--scale images at the various levels of a pyramid), which is essential for an effective monitoring of multiresolution and/or multilevel algorithms.

A considerable effort has been devoted to the realization of an environment where the user can make familiarity with the machine capabilities and is encouraged to test them with an immediate feedback. We provide an overview of the whole development environment (sections 3 to 6), after a very short hint about PAPIA2 flat pyramid architecture, given in section 2. The interaction among the modules which are directly devoted to the debugging process is explained in section 7; finally section 8 shows the way this environment can be modified to suit some other architectures.

2: The flat pyramid

PAPIA2 is a "virtual" pyramid [5,6] embedded in a square array of bit-serial processors by means of a recursively definable projection of the upper-layer PE's onto the pyramid base (Fig. 1). The array can work either in SIMD

0-8186-2925-8/92 $3.00 © 1992 IEEE

mode when operating as the base of the pyramid or in Multi--SIMDgrid fashion when more layers. above the base (grids) are simultaneously active with horizontal communications In the latter case physical adjacency of logically 4--connected PE's is achieved by exploiting short-porting capabilities of PE's and multiple configurations of the switch--lattice that mediates PE-to-PE links. This way, interconnections are highly simplified when compared to other physical or simulated pyramidal solutions, while high integrability and extended modularity may be achieved and popular fault-tolerant strategies for bidimensional arrays of homogeneous PE's still prove useful. Another great advantage of this architecture is the ability to support both standard mesh array processing and efficient multiresolution algorithms through 3 different operating modes: as the pyramid base, as the upper-layers with parent-child communication, as the upper-layers with near-neighbour communication. A particular RISC-like design of the processor ensemble has allowed to cram all the functionalities in a compact yet somewhat complex set of 20-bit-long highly structured opcodes.

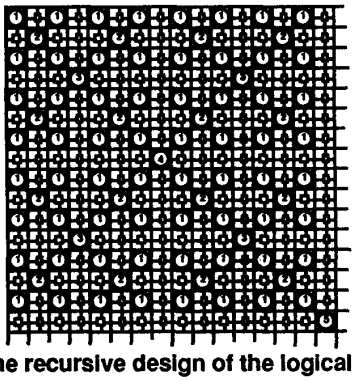

Fig. 1: The recursive design of the logical pyramid.

3: The software environment

Designing a programming environment from scratch is always a demanding task: choices have to be made about the closeness to machine details and peculiarities, the tools to be provided, the type of user-interface, and so on. A number of reasons concurred to select a low-level simulator as the natural (and necessary) starting point for a long winded project that foresees the creation of a high-level counterpart of the environment. The addition of a graphical front-end, made up by three tools - a "visual assembler", a debugger and a PAPIA2-status visualizer - completed the project design and structure [7] (Fig. 2).

A simulator that highly matches the hardware specs is required since, up to now, no physical prototype exists. The benefit granted by the low-level approach is twofold: on the hardware side it is the only one able to supply reliable performance figures to justify design options, whereas for the software it represents a valuable testbed for the chosen instruction-set. The graphical user-interface is obviously a must when dealing with image-processing hardware, but it would be an oversimplification limiting the rationale to that: it is well known that graphical representation of data is more effective than a textual one in transmitting to the user the extremely high volume of informations produced during the execution of concurrent programs[8]; also, the widespread acceptance of direct-manipulation paradigms [9] is the undoubtful sign that carefully designed WIMP (Windows-Icons-Mouse-PopUpMenues) applications [10] lead to higher efficiency. Furthermore, the availability of a de-facto standard like the X-Window System [11] gives the environment a significant portability .

4: *PySim*, the simulator

The nucleus of the programming environment for PAPIA2 is a simulator of the machine described in the previous sections. This tool, written entirely in C and running on a Unix workstation, is capable of simulating the execution of PAPIA2 machine instructions on arrays whose size is limited only by the addressable memory-range of a Unix process. The execution of sequences of instructions may be carried out in a SIMD or in a multi-SIMD fashion. In the latter case, the granularity of the distribution of different instructions across the array can be controlled by the user and may vary from a single instruction for all the PE's (falling thus back to a pure SIMD mode) to a different instruction for each plane of the pyramid.

The array simulation is not the only task this program must perform: I/O and machine management functions must be provided; furthermore the support for a transparent monitoring of instruction execution and PE status evolution should be given. These three main activity-areas offer a functional guideline to group all the different tasks, thus making the logical structure of PySim higly modular: every action is implemented by a command and correlated commands are grouped into distict sections.

5: *XILEBO*, a visual instruction assembler

It is quite clear that the usual assembler approach is inadequate to express the variety of different features PAPIA2 offers: the need of stating operating mode and direction of communication to complete the basic instruction-mnemonic would have required quite verbose, annoyingly long statements. Thus Xilebo (namely X Interface for the LEarning of Basic Operations) was devised to provide programmers a workbench (a window) where they can "build" up instructions to be executed by the PySim simulator and experiment their effects.

As a primary benefit the user is not requested to memorize the whole instructions set: a scroll-list allows the selection of an instruction among the ones associated to the functional blocks the programmer highlights, further selectable icons permit the specification of the operational mode, of memory addresses, planes and neighbours involved. Secondly, a list of the available directives is received from

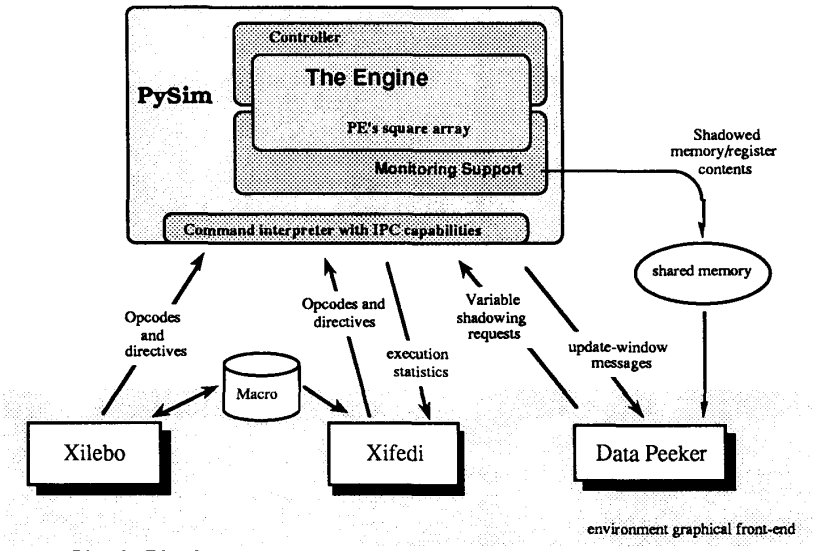

Fig. 2: Block diagram of the PAPIA2 development environment

PySim at start-up and is used to define a pop-up menu whose items are selectable by the user whenever one of such directives is needed. A "macro editor" is also included in the workbench for the recording and the playing-back of sequences of instructions.

Trough a human-computer interaction that is simple enough for novices yet fast enough for experts, users are facilitated in exploring and understanding system capabilities, while instructions correctness is automatically ensured (and typing effort minimized). Moreover, immediate evaluation of the instructions gives a fruitful interactive flavour.

6: The *Data Peeker* and *Xifedi*

The design of Xifedi (X-Interface For Enhanced Debugging Interaction) is more an evolutionary one: by means of an easy-to-use interface (Fig. 3) the user can control the execution of programs written in PAPIA2 microcode by running the program, single stepping through the code, setting or removing breakpoints.

Machine dependent knowledge is very limited: every non-empty line, not beginning with a dash symbol, is taken to be a working line and is passed to PySim. After the execution of this instruction/directive PySim communicates to Xifedi which group the instruction belonged to: normal, recursive, I/O, other; so that final statistics can be evaluated.

Monitoring of every action carried out by PySim is performed by the Data Peeker (Fig. 4). This tool allows the visualization of registers, memory locations and memory segments to handle grey-scale cases. Two display capabilities are available: the array mode shows the whole PE's array as a bitmap/pixmap, taking advantage of the one-to-one mapping between the array and the image being processed; the pyramid mode shows one image for each plane, halving the resolution (i.e. keeping constant the dimensions) when moving from the base upwards. The list of shown items can be varied at run time: consistently with the way Xilebo handles the directives provided by PySim, the Data Peeker shows the displayable items in a menu and offers a command button to 'undisplay' unwanted images.

7: Debugging programs: an example

Even if only a small set of capabilities is built in the debugger itself, its interaction with the Data Peeker and Xilebo places it in a class of its own: it is quite handy watching the evolution of an algorithm on the array, with the possibility of stopping it and issuing instructions to the machine if some additional steps are required!

As a hint about this environment we show the Data Peeker display during a simulation. In this case a binary image labeling procedure based on a heat-diffusion process is used. After assigning to all the contour elements an arbitrary integer value which initializes the process, a simulation of heat diffusion from the contour towards the interior of every connected component is run. At each iteration of the diffusion step, new pixels will be contaminated provided they lie in the interior of the object. After a number of steps the contour elements will preserve high values corresponding to local convexities and will produce a sharp decrement in value corresponding to local concavities. Since concavities and convexities may have a wide range of average curvature for a given diffusion coefficient, and the number of iteration steps producing a meaningful value indicating a concavity (or a convexity) will vary within a wide range (a small number of iterations for a local concavity (convexity) and a large number for a smooth concavity (convexity)), the different concavities (convexities) will be detected after a different number of iteration steps. This is why a pyramidal, multiresolution architecture is best suited to handle this al-

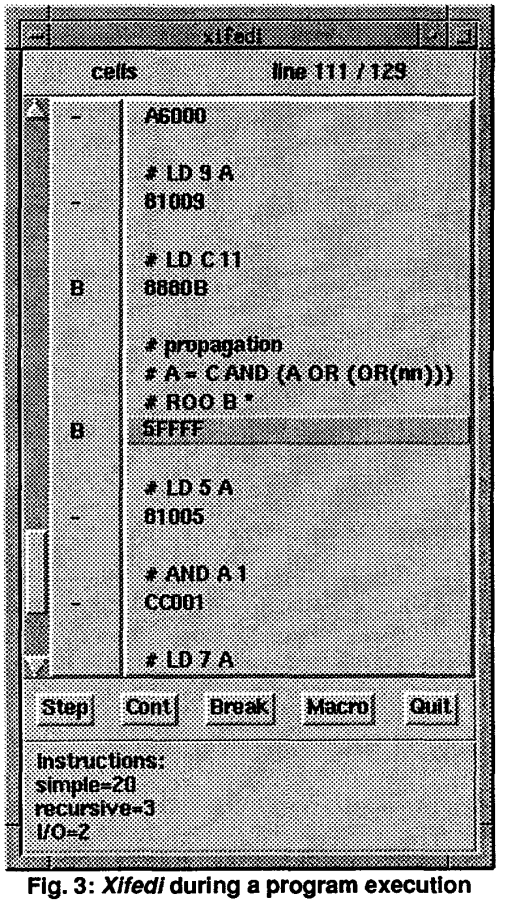

Fig. 3: *Xifedi* during a program execution

gorithm. Figures 4 and 5 exemplify the way the user can visually follow this process: the starting image is depicted in Fig. 4-b while Fig. 4-c shows its lower resolution versions at the levels 1, 2, 3 and 4 of the pyramid; the same images after 50 iterations of the diffusion process are illustrated in Fig. 5.

8: Toward machine independence

Starting from the problem of code reusability, the modular structure of the environment was decided in order to shift toward a component-based software development at two different levels: a basic one pertaining to sources and a more abstract one related to the different modules which were programmed. What appeared immediately was the possibility to achieve machine independence of Data Peeker and Xifedi by adding a further layer - a protocol, between them and the simulator.

In fact there are only a few rules that any simulator, such as PySim, must follow so that it can successfully interface with Xifedi and the Data Peeker. As long as Xifedi is concerned only the ability to perform a single instruction/directive is required, being the machine-dependent part into the executed program itself. On the other hand the Data Peeker protocol asks for a continuous communication between the two programs, given the client-server approach we used. The tasks that the simulator is requested to perform are: to send a list of "objects" that can be displayed, to transform every (machine-dependent) "object" into a standard, predefined format, and to communicate the completion of an instruction so that the Data Peeker can update the display.

9: Summary

We presented a modular, portable and graphical software-development system built for the PAPIA2 pyramidal machine as a paradigm of a novel, visual approach to program development and debugging. Special importance has been attached to the representation of data at the required level of abstraction, so that the visual nature of the debugging process is strongly supported.

References

1. C.D. Norton and E.P. Glinert, "A Visual Environment for Designing and Simulating Execution of Processor Arrays", *Proc. 10th Int. Conf. on Patt. Rec.*, 1990.
2. S. Dacic, J.M. Frecaut and B. Zavidovique, "Software Environment For Complex Machine Programming", *Proc. COMPEURO 90*, May 1990.
3. A.L. Amber and M.M. Burnett, "Influence of Visual Technology on the Evolution of Language Environments", *Computer*, Vol. 22, No. 10, Oct. 1989.
4. *IEEE Software*, May 91, Cover Articles.
5. V. Cantoni and S. Levialdi, "Multiprocessor Computing for Images", *Procedings of the IEEE*, Vol. 76, No. 8, Aug. 1988.
6. M.G. Albanesi, V. Cantoni, U. Cei, M. Ferretti and M. Mosconi, "Embedding Pyramids into Mesh Arrays", in H. Li, Q. F. Stout (eds.) *Reconfigurable Massively Parallel Computers*, Prentice-Hall, April 1990.
7. A. Biancardi, V. Cantoni, U. Cei and M. Mosconi, "Program Development and Coding on a Fine Grained Vision Machine", tbp on Machine Vision and Applications.
8. G.-C. Roman and K. C. Cox, "A Declarative Approach to Visualizing Concurrent Computations", *Computer*, Vol. 22, No. 10, Oct. 1989.
9. B. Shneiderman, "Designing the User Interface", Addison-Wesley, 1987.
10. D. A. Norman, "Design Priciples for Human-Computer Interfaces", *Proc. of CHI '83*, 1983.
11. R. Scheifler and J. Gettys, *The X Window System, Second Edition*, DEC Press, 1990.

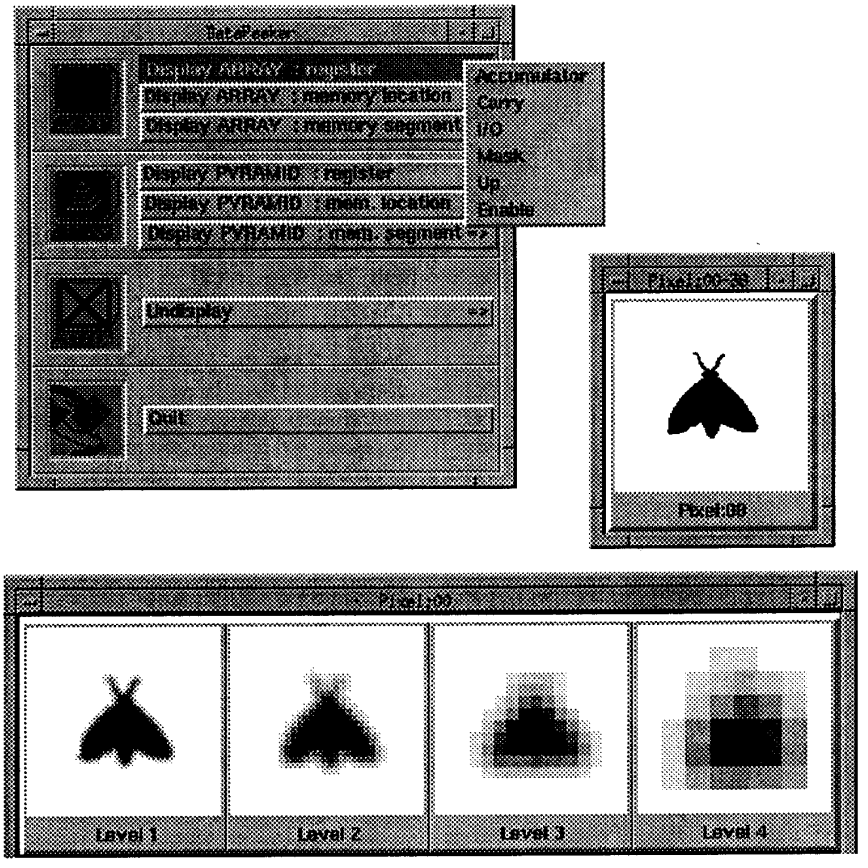

Fig.4 The Data Peeker main window (a) and two of its monitor windows showing, in the form of grey-scale images, the contents of a memory segment of a 128x128 array (base, memory locations 08-0F) (b), and of the four upper layers (memory locations 00-07) (c).

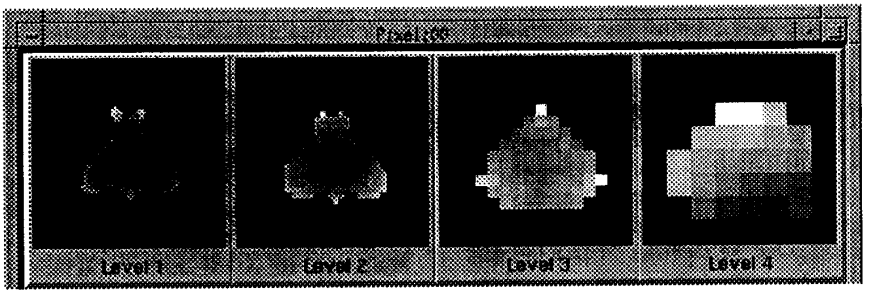

Fig.5 The Data Peeker monitor window of Fig. 4-c as it appears during the diffusion example.