SYSTOOL – SYSTOLIC ARRAY PROGRAMMING TOOL

1.5.5

Baruch Nissenbaum and Yair Be'ery

Department of Electronic Communications, Control and Computer Systems,
Tel–Aviv University, Tel Aviv 69978, Israel.

ABSTRACT

Programmable systolic arrays (PSA) are one of the promising parallel processor architectures. SYSTOOL combines the power of the C language, together with a powerful simulator to form a state of art integrated environment for systolic array programming. In this paper we shall review the PSA concept, present the simulated processors and describe the structure of SYSTOOL and its features.

1. INTRODUCTION

Programmable systolic array [1] is a SIMD computer composed of an array of processing elements (PE's). Each PE is a simple (usually one bit) processor containing a few registers, RAM, and an ALU. The most popular PSA is built from one rectangular array, in which every PE can communicate with its four nearest neighbors. PE's on the boundary of the array may be connected to I/O pins of the chip, providing communication to the outside world (edge communication). All PE's in the array execute the same command in parallel.

PSA may have very high processing power. For example the ASAP processor (described below) may do over 5,000,000,000 8–bit multiplications per second (total), or over $20 \cdot 10^9$ 16 bit additions per second. This performance is possible because of the parallel architecture of the PSA and the large number of processors in the array.

Because of the simple structure of every PE, the repetitive nature of the processing array and the simple communication paths, it is relatively easy to pack many PE's into one silicon chip (or wafer). It is much easier to design a large systolic array chip than to design a conventional ALU of the same size. Array structure may be cascadeable, and array size may grow as problem size increases. It is possible to build large arrays by a simple connection of several chips each containing a smaller array.

The bottleneck preventing the wide use of the full power of PSA's is the lack of software. Specialized algorithms are needed to make the most of this none–standard architecture. Software development for PSA's is hard since programming tools are very few, and the available programming tools are very limited. Existing tools are mostly tailored around a specific hardware circuit and do not support more general needs. There is no real high level language compiler for systolic arrays (yet).

The most important contribute of SYSTOOL is to introduce modern software development methods to the PSA field. SYSTOOL (running on an IBM/PC or a compatible computer) helps creating software for PSA's, both for educational purpose and for practical software development. SYSTOOL can be used at early stages of a project, to help in determining the feasibility of systolic arrays to the problem at hand, without large investments in hardware. The authors believe that SYSTOOL has an important role in making PSA concept more popular and widely used.

2. DESCRIPTION OF THE TARGET PROCESSORS

SYSTOOL currently supports two PSA chips: the NCR GAPP and the ESPRIT project ASAP (mission 824B). GAPP is a cascadeable 12x6 PE array (per chip), running at 10 Mhz [2], available as a commercial product of NCR since 1984. ASAP in a WSI (wafer scale integration) chip containing a 128x128 PE array, with 30 Mhz clock [3]. ASAP is now at final stage of its development. Most of the following description will concentrate on the the newer processor.

Each one of the ASAP 16384 processing elements contains (see Fig. 1) 4 one–bit registers, two segments of RAM with 64 bit each, one-bit ALU, and a flag register. Register to register assignments, communication, RAM and ALU operations may be performed simultaneously during the same clock cycle. PE operation may be disabled according to the flag register content. Array operation is determined by a 17–bit control word and two 5–bit address words. Edge PE bits are available at the chip boundary (128·5 I/O bits). Global–output bit is available, so global operations can be performed very fast (eg. find the maximum value in the array).

3. DESCRIPTION OF SYSTOOL

3.1 General

A PSA simulator must simulate two separate components of the hardware: the PSA itself and the program sequencer. Program sequencer can be very simple (counter + ROM) or very complex (custom made VLSI micro controller). SYSTOOL gives the user a systolic command translator and a powerful PSA simulator (see Fig. 2). In addition SYSTOOL uses the power of a commercial C compiler (Turbo–C) for its user interface, and the C language is used to simulate the control mechanism of the PSA. SYSTOOL gives a possibility to simulate edge interconnection, external logic connected to the edges and I/O streams, thus making the simulation closer to real life. Extensive error report is supplied by both the commercial compiler and the systolic simulator.

3.2 SYSTOOL structure & operation

For generating a PSA program, a C program is written by the user with the Turbo–C editor (referred as 'user program'). This program is compiled, and then linked to the SYSTOOL library. The result of the compilation is an executable program (user–program block in Fig. 2). While running, the user program activates an array–command translator via function calls. The translator translates PE commands written in string format to an intermediate data structure. All translated commands are tested for assignment errors, communication errors, and other processor–dependent errors. The error report contains description of the error, a copy of the command and a pointer to the offensive word, so the user can pin point the problem. PE memory management mechanism is also implemented in this stage.

The array–simulator interprets commands from the translator and applies the commands to the array data stored in the RAM. Direct control of the simulator from user program is also possible. By using direct simulator control the user can define the size of the simulated array, get printouts of register and memory content or read them from files, and activate the debugging facilities of the simulator. The simulator is able to access the DOS file system, execute other programs and open a new DOS shell. Trace mode, breakpoints and cycle counter are some of the debugging tool supported by the SYSTOOL simulator module.

3.3 Special features of SYSTOOL

PE memory management

The GAPP has 128 bits of RAM (for each PE) organized as one address space. The 128 RAM bits of the ASAP are split into two segments of 64 bits each. The two segments of ASAP RAM can be addressed simultaneously in the same command (cycle). SYSTOOL's unique memory management system enable the users of both processors to access RAM naturally, by using 'image' data type.

Image is a 2 dimension array of values, where each PE holds one such value in its RAM. Image dimensions equal the dimensions of the PE array, yet the number of bits per value (image size or depth) is user definable. Images may be declared, allocated and freed anywhere in the user source code. Allocation is dynamic; PE RAM space is allocated from the available free memory and, after use, may be returned to the free memory pool. The whole memory management is controlled automatically by the SYSTOOL system. Since memory management is done at translation time, this process do not degrade the speed of a PSA controlled by a simple sequencer.

Special C macros are defined for printout of RAM images and for reading RAM images from files. These macros only need the image name as an input. The user does not need to know the actual address of the image in PE RAM space. Care was taken to ensure that image I/O will not cause distortion to image data in case of different array dimensions used for read and write.

Users can simulate a program module from a pre–defined state and print images at any step during program execution, by using friendly image I/O facilities. Printout of images is a very important debugging tool, and is used to validate or to demonstrate execution of algorithms. Images can be printed or read in decimal, octal, hexadecimal or user–defined format.

Edge communication

In actual systems the PSA must be connected to input and output data paths. There is a very large number of ways to connect a PSA to the surrounding circuit. The connections have a great impact on the algorithms and on overall system performance. Some connection possibilities appear in Fig. 3.

A relatively new feature of SYSTOOL simulator is the possibility to define edge connections. SYSTOOL provides three options for connecting edge I/O pins, which correspond to practical system connections:

1. Every edge can be connected to every other edge on the array boundary or to a constant logic state. Cyclic, shift and corner–turn are some of the possible connections.
2. Edge I/O is possible from a data file to any list of edge pins, or from an edge list to file. I/O data streams can be simulated in this way.
3. Edges can be 'connected' to C variables, so any external logic may be simulated by the user program. The control flow of the user program can be dependent on the edge data.

It is possible to mix all of the above options in a single simulation. Connections may be re–defined during simulation, thus simulation of external switching of logic is possible. Connection definition is easy to use, it has a simple syntax, and it may be activated from the interactive environment.

Library usage

Since systolic programs are written in C, C functions can be used to write systolic functions. These functions can be compiled separately and linked to the main program in a later stage. Libraries of systolic function can be created as well. Standard libraries may include general algorithms for various applications such as image addition, multiplication, 3x3 convolution, sort, floating point operations, etc.

Using the image system, functions may be general – independent of the actual allocation of images in RAM and independent on argument size. For example 8–bit addition and 32–bit addition may be performed using the same function.

Simplified syntax

A simple syntax was defined for PE code description. All commands are simple assignment commands. Special pseudo registers were defined (by which communication, ALU operations, and constants are handled naturally) thus contribute to the readability of the programs. In the case of ASAP ALU, or ASAP BIT input [3] the use of pseudo registers may save a lot of the PE instruction length. Appropriate error messages were added to handle the functions of these pseudo registers.

Interactive simulation mode

Although most simulation directives are given in the user program, there exists an interactive simulation mode. In this mode the user can give the simulator commands one by one, examine register and RAM content and activate most of the simulator options. Interactive mode may be entered from normal simulation, and execution can then be transferred back to normal simulation. The interactive mode is very useful for debugging of programs.

PE structure definition

New PE designs (close in nature to ASAP & GAPP) may be declared for simulation. This option is useful for developing software for a new systolic array, or in a research for improved PE architecture. Structure definition is done interactively using a separate program. Simulator speed is not degraded by this option.

Direct simulator commands

A unique feature of SYSTOOL is its ability to control the simulator by commands included in the user programs. This option enables the user to add remarks, format the final output, activate or deactivate debugging, activate external programs using DOS and much more. Explicit simulation setup is not necessary on every simulation, as it is with other simulators.

4. PERFORMANCE OF SYSTOOL SIMULATOR

The performance of a programming environment such as SYSTOOL is composed of a few factors, all contribute to the process of a program development:

1. Speed of simulation.
2. Programming cycle time.
3. Debugging facilities.
4. Ease of use (human interface) and learning effort.

Simulation speed measurements of SYSTOOL gave very good results. Measurement method and the results are elaborated below.

Programming cycle time is the time it takes from changing the source program, to the time that simulation results are available (including: C–compilation, PE–code translation & simulation). This time is in the order of 10–30 seconds (for small–medium programs). No editor loading or compiler loading time is necessary, since the Turbo–C environment is used.

As mentioned, PE code debugging is done at simulation time. About 30 different error messages are available by SYSTOOL. Debugging modes such as trace and break are supported. A special interactive mode is available for more powerful debugging. Since simulator directives are inserted in the user source code, the user is relived from the need to repeatedly setup simulation interactively as is the case with other simulators.

User convenience is harder to measure, yet all users of SYSTOOL spent very short learning time (few hours) and all users noted that SYSTOOL was friendly and powerful yet very easy to used compared to another GAPP simulator.

Evaluating simulator performance

As can be seen from the system structure, three modules take part in the actual simulation: the user C program,the PE (GAPP or ASAP) command translator & memory manager and the array simulator. The dominant module is the array simulator but at rare cases the user program can be very complex and consume a lot of time.

Simulation time depends mainly on the array size and the complexity of the commands. Command complexity can range from very simple – one assignment per command, to very complex command – using many assignments per command which activate memory references, communication, ALU and flag usage.

In the evaluation, three typical programs were tested: The first is the assignment test, it is a loop of 400 simple assignment commands. It was executed on a 32x32 array. This measure give peak performance of the simulator. The second test is a mix of additions and n–bit shift operations. Here there were more then one assignment per command, and communication was used. Since the commands are of average complexity, the time is a typical simulation time for typical programs. The third test is an image multiply test. Here, two images are multiplied. Commands are complex and there is an intensive use of conditional operations in the algorithm. Execution time is typical for a highly optimized code.

In all cases the number of cycles were counted, and the execution time was normalized to be the time per complex command per a single PE in the array. This measure includes translation time and user–program execution time overhead. All tests were made on a PC–AT compatible (10Mhz). The results are summarized in Table 1.

The speed of another published ASAP simulation [4] is 4 mili–second per PE. Thus SYSTOOL is about 400 to 2300 times faster. SYSTOOL running on an IBM–PC was reported to be faster then the NCR GAL [5] running on a VAX computer, however, exact measurement was not made.

Overall performance of the programming cycle depend also on compilation speed of C. The use of Turbo–C for compilation gives compilation time of less then 10 seconds (about 2–4 seconds if using RAM disk). This compilation speed is far better then any of the other known systolic array simulators and contributes to its superior performance.

6. CONCLUSIONS

SYSTOOL was first developed for the GAPP in February 88. It was later generalized and modified for the ASAP processor. A final ASAP version that used the C environment and was considerably faster then the previous version was finished in April 88. This version was used, in Tel–Aviv University, by students of a graduate course for their final work. Since then, a user defined output was added, to support floating point library development, edge communication was added, and a new module for definition of PE architecture is in its final stage of development. SYSTOOL is currently utilized for developing a software library for the ASAP processor and is used as a primary training tool for teaching the PSA concept at Tel–Aviv University.

REFERENCES

[1] S. Y. Kung, VLSI ARRAY PROCESSORS, Prentice Hall, Englewood Cliffs, NJ, 1988.

[2] R. Davis and D. Thomas, "Systolic Array Chip Matches the Pace of High–Speed Processing", Electronic Design, Oct. 31St 1984, pp. 207–218, See also Nov. 15, Nov 29, Dec. 13 1984, Jan. 10 1985.

[3] J. Trilhe and G. Saucier, "WSI– The Challenge of the Future," Comp–Euro 87, 1'st Int'l Conf. On Computer Technology, Systems and Applications, Hamburg, Germany, May 11th – 15th, pp. 531–541.

[4] R. Wickerath, H. Fridrich, M. Huch, and M. Glesner, "Description of the simulator for systolic array parallel processors SAPPhire," Research Report, Technische Hochschule Dramstadt, Institute f Halbeleitertechnic, West Germany, June 1987.

[5] NCR, "GAPP Algorithm Language – GAL," Manual, Fort Collins, Colorado, 1986.

Fig. 1 ASAP Architecture

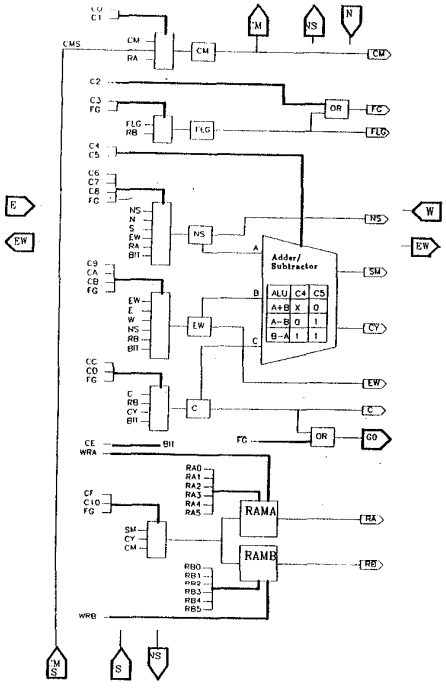

Fig. 2 SYSTOOL schematic structure

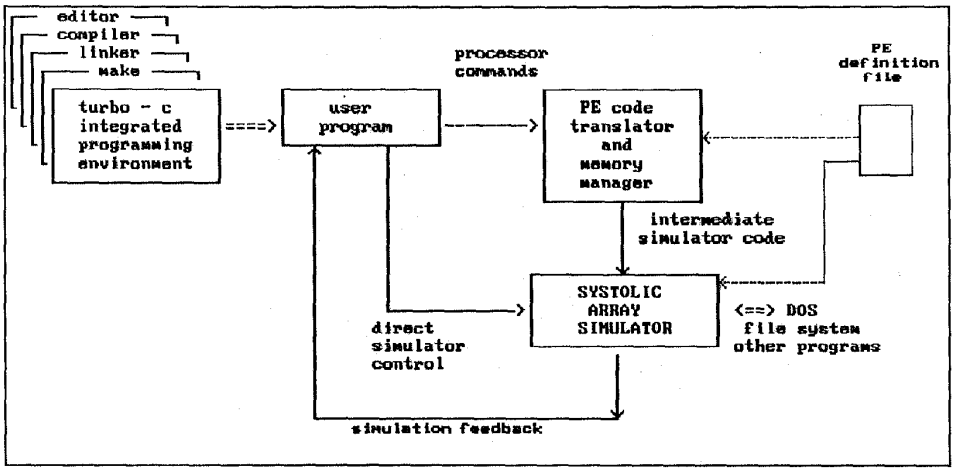

Table 1. SYSTOOL performance

TEST	ARRAY SIZE	NUMBER OF PE COMMANDS	TOTAL TIME	AVERAGE TIME PER PE COMMAND
Single assignment	32x32	400	0.74 sec	1.7 micro sec
Add image shift image mix	32x32	282	1.54 sec	5.3 micro sec
Image multiply	64x32	88	1.76 sec	9.8 micro sec

Fig. 4 Sample program

```
/* test addition of integer to image */

main()
{
   image a;                              /*  define a as an image     */
   sim("xy 16 8");                       /*  set array size to 16x8   */
   a=img_alloc(RA,8 );                   /*  allocate 8 bits for a    */
   sim_read("im_a",a);                   /*  read data from file      */
                                         /*   "im_a" into image a     */
   add_int_to_ra(a,40);                  /*  activate the function    */
   sim_print(a);                         /*  print the reasults       */
}


#define bit(x,i) ((x)>>(i) & 1)          /*  test bit i in integer x   */

add_int_to_ra(image a, int x)
/*   add an integer x to an image a,  a is of arbitrary size (depth)     */
{
   int i,j;
   for(i=0; i<size(a) && bit(x,i)==0; i++); /* find first "1" bit in x   */
   if(i==size(a))                        /* if nothing too add ...     */
      return;                            /* then leave the function    */
   PE("c=0 ns=0 ew=0");                  /* clear carry                */
   for(; i<size(a); i++) {
      if( bit(x,i)==1 ) {                /* if the bit to add is 1 ..  */
         PE("c=cy ns=ra ew=1 ra=sm",a,i); /* add 1 to bit i of image a */
      }
      else  {
         PE("c=cy ns=ra ew=0 ra=sm",a,i); /* else: add 0 to bit i      */
      }
   }
}
```

Fig. 3 Edge–communication examples