Towards a Mathematical Specification of the APE100 Architecture: The APESE Model*

Egon Börger [a], Giuseppe Del Castillo [a], Paola Glavan [a], Dean Rosenzweig [b]

[a]Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy.
boerger,delcasti,glavan@di.unipi.it

[b]FSB, University of Zagreb, Salajeva 5, 41000 Zagreb, Croatia.
dean@cromath.math.hr

This paper provides the first step of a full mathematical description of the APE100 parallel architecture. The description consists of several models, at different levels of abstraction, corresponding to views of the architecture provided by different languages used within the APE100 *compilation chain* (a crucial part of the software environment of APE100).

Here we present the *primary model*, based on the relevant subset of APESE, a high level language specially designed for APE100 and constituting the source language in the compilation chain: APESE reflects closely the APE100 model of parallel execution and is therefore adequate for an abstract description of the main features of the latter. Stepwise refinement will lead us from the APESE model of this paper to the hardware level of APE100, using evolving algebras as specification method (see [1,2]).

APE100 provides a beautiful example of an evolving algebra which models lock-step parallelism: at each step all rules which can be fired are fired simultaneously.

Keyword Codes: C.1; C.3; J.2
Keywords: Processor Architectures; Special–Purpose and Application–Based Systems; Physical Sciences and Engineering

Introduction

The APE100 parallel processor has been developed by a group of physicists in Pisa and Rome as a dedicated machine for floating point intensive scientific applications, in particular numerical simulation in Lattice Gauge Theory (see [3,4]).

The APE100 architecture is of type SIMD, therefore it realises a lock-step model of parallelism. It consists of a master *controller* (called zCPU, with a store for data and a store for the program), which controls a large number n of isomorphic processing nodes (*floating point units*, each with its own local data store). The floating point units (FPU's) run in lock-step, steered by the controller, which controls the (sequential) instruction flow and performs all integer arithmetic operations involved in the program, in particular

*In: Proc. *IFIP Congress* '94 (13th World Computer Congress), North-Holland Pu.Co.

the evaluation of addresses for the floating point units (the isomorphy implies that these addresses form an address space which is common to all FPU's). Thus, all FPU's, although acting on possibly different data, execute at each step the same instruction. Nevertheless, a rudimentary form of local program conditioning is provided: modifications of local data are subject to locally tested conditions and not effected by those FPU's where these conditions are false. Thus, each FPU i has its own stack IF_i (so called "IF-stack") to store the values of nested local conditions.

The processing nodes are topologically layed out as a three-dimensional torus, allowing each FPU i to access its own store sto_i and the store $sto_{neighb_j(i)}$ of its j-th neighbour, $j = 1, \ldots, 6$.

A simple high level programming language, APESE, has been developed for the APE100 family of parallel computers, to ease the transition ¿from standard scientific programming to parallel programming. The basic instructions are (integer and floating point) assignment and (global and local) conditionals, enriched by other usual (FORTRAN style) constructs, which we skip here because their inclusion into our treatment is routine.

The controller executes integer assignments only, whereas the FPU's execute only floating point assignments; the controller's (global) conditions check what is to be done (manipulation of instructions and memory addresses, for both the controller itself and the local nodes), the local conditions establish where local assignments are to take effect.

The APESE Model

The rules of the evolving algebra we are going to define can be read and understood as pseudo-code over abstract data. The reader who wants to know more about the precise definition and the foundations of evolving algebras is refered to the paper by Gurevich in this volume.

Our evolving algebra reflects the APE100 model of parallel execution, as viewed by the APESE programmer. To let the lock-step parallelism of APE100 stand out explicitly through evolving algebra rules, we assume APESE programs to be layed out ("compiled") in the usual manner into a flowchart of *NODE*s decorated (via a function *cmd*) by atomic actions and linked by edges (expressed using functions *next*, *yes*, *no*). The (global) control is expressed by a dynamically updated distinguished element *curr_node* ("program counter"), which we like to see through the metaphor of a daemon walking through the flowchart and doing the atomic actions attached to the nodes by firing at each step a maximal consistent set of rules.

Walking is formalized by updates $curr\_node := f(curr\_node)$, written "*do f*", where f is *next*, *yes* or *no*; the required *action* is expressed by "doing *action*", which stands for $cmd(curr\_node) = action$.

In the nodes of the APESE flowchart, five types of atomic actions can appear: global and local assignments, global conditions representing conditionals, local conditions decorated with **where**, **endwhere**. The semantics of each type of action is expressed by a corresponding group of evolving algebra rules.

In order to abstract from details of expression evaluation in a given environment and store, bindings of identifiers are abstractly represented by a distinguished element *curr_env* (current environment), which is a static function here, since the relevant subset of APESE

doesn't contain procedures. The memories of the FPU's, which are all accessed from the same address space, are realised using distinct local stores $sto_1, \ldots, sto_n$, which associate n local values to each location; the controller memory is represented by sto_0 (see the Appendix for details on stores, evaluation and address calculation). We are now ready to describe the rules for the five atomic actions.

If the daemon is *doing* an integer assignment (of the form $j = exp$, where j is a variable or an array element and exp is an expression), the function exp_ev will evaluate expression exp and the function $address$ will calculate the address of j, using $curr\_env$ and the controller store sto_0. The controller store sto_0 will then be updated, by writing the value $exp\_ev(exp, curr\_env, sto_0)$ to the location denoted by $address(j, curr\_env, sto_0)$ in sto_0 (see Appendix for details of evaluation, address calculation and writing).

if doing $j = exp$
then write $exp\_ev(exp, curr\_env, sto_0)$ to $address(j, curr\_env, sto_0)$ in sto_0
do next

When executing a global logical condition $global\_cond$, the controller evaluates the condition (using a function $cond\_ev$) and moves on according to the computed value. This evaluation may need all the stores, since a global condition may contain a quantified local condition (see the Appendix).

if doing $global\_cond$
then if $cond\_ev(global\_cond, curr\_env, \langle sto_0, sto_1, \ldots, sto_n \rangle)$ **then** *do yes* **else** *do no*

After each operation the "program counter" must be updated, i.e. the daemon has to move. Since floating point units cannot access the program counter, this update must be done by the controller. Hence a final controller rule:

if doing $local\_op$ **then** *do next*

where $local\_op \in \{x = exp,$ **where** $local\_cond,$ **endwhere**$\}$ is the set of local operations, i.e. operations carried out by the FPU's (and modeled by the FPU rules given below). Since at each moment a maximal consistent set of evolving algebra rules is to be applied, this rule will be executed together with the set of local rules for op, one for each FPU. Local rules come, namely, with an index $i \in FPU = \{1, \ldots, n\}$, and stand (as rule schemata) for n rules, one for each FPU. They are defined as follows.

An FPU can execute the following actions: **where** $local\_cond$, **endwhere** and local assignment. If the (local) logical condition of a **where** statement is satisfied, floating point assignments in the scope of that **where**, as given by the flowchart, are executed — otherwise they can be seen, at this level of abstraction, as skipped. This construct is modeled using IF_i, the IF-stack of floating point unit i, which grows or shrinks each time **where** or **endwhere** is executed.[2]

if doing **where** $local\_cond$ **then** $IF_i := \langle cond\_ev(local\_cond, curr\_env, sto_i) \mid IF_i \rangle$
if doing **endwhere** $\wedge$ $IF_i = \langle b \mid rest \rangle$ **then** $IF_i := rest$

[2]The IF-stacks are implemented in hardware as circular bit sequences of length 8; since the APESE compiler accepts only programs with at most 8 nested **where**s, this implementation works correctly as a stack.

Local assignments are formalised using the same functions as in global assignment. The only difference is that the expression is evaluated in the local store sto_i, and that the execution is conditioned by the conjunction of all bits in IF_i.

if doing $x = exp \ \wedge \ (\bigwedge_{b \in IF_i} b)$
then write $exp\_ev(exp, curr\_env, sto_i)$ to $address(x, curr\_env, sto_0)$ in sto_i

Remark 1 In view of the memory mapping feature, it may happen that it is a neighbouring store which in fact gets updated by the above rule (see the Appendix).
Remark 2 Since, at this level of abstraction, address calculation is represented by the static function *address* (see the Appendix), the fact that all address calculations are performed by the controller is reflected in the rule by applying *address* to controller memory sto_0 only.

The *initial states* (static algebras), starting from which we are going to apply the evolving algebra rules given above, are required to satisfy the following conditions:
– the given (correct) APESE program is layed out as flowchart;
– *curr_node* is set to the flowchart node corresponding to the first statement to be executed;
– the declarations of the program are represented by *curr_env*;
– all IF-stacks are initialized to true (i.e. for each $i \in \{1, \ldots, n\}$, $IF_i = \langle \mathbf{true} \rangle$);
– the stores are initialized to store the given data.

Appendix: Statics of APESE

Expressions and Values — Binding, Storage, Evaluation

In order to explain the peculiar memory mapping feature, we need to elaborate expressions, values and storage organization.

Expressions of APESE are represented in a universe *EXP* and evaluated by a function *exp_ev*. We leave the expression syntax abstract, assuming that the basic components of expressions are *constants* (integer or floating point values, taken from a universe $VAL = INTEGER \cup FLOAT$ and denoted by the metavariable α), *variables* (x is used to denote a variable identifier) and *array elements* (of the form $a[exp_1, \ldots, exp_m]$, where a denotes an array identifier and the exp_j are expressions).

Association of identifiers and objects is abstractly represented in a universe *ENV* of *environments*, and accessed by a partial function *bind*, defined on $ID \times ENV$. Variable identifiers are bound to memory *locations*, while bindings of array identifiers contain extra information about dimensions, required for address computations (i.e. $bind(x, env) \in LOC$, $bind(a, env) \in LOC \times \mathbf{N}^*$). A distinguished element $curr\_env \in ENV$ denotes the current environment.

Association of locations and values is abstractly represented in a universe *STORE* of memory states, and accessed by a partial function $cont : LOC \times STORE \rightharpoonup VAL$ ("fetching the content of a location in a given store"). Writing value α to location *loc* in *sto* can now be defined as modifying *sto* to $sto\{\alpha/loc\}$, where the latter is defined by the equation

$$cont(loc_1, sto\{\alpha/loc_2\}) = \begin{cases} \alpha & \text{if } loc_1 = loc_2 \\ cont(loc_1, sto) & \text{otherwise.} \end{cases}$$

Since in APE100 the (global) data store of the controller (containing integer values) is different from the (local) data stores of the FPU's (containing floating point values), we use two types of store, $STORE_g$ (containing sto_0) and $STORE_l$ (containing sto_i for $i > 0$), such that $STORE = STORE_g + STORE_l$, and two corresponding types of locations, LOC_g and LOC_l, such that $LOC = LOC_g + LOC_l$, with

$$\begin{array}{lll} cont(loc, sto) & \in & INTEGER \quad \text{if } loc \in LOC_g \text{ and } sto \in STORE_g \\ cont(loc, sto) & \in & FLOAT \quad\;\; \text{if } loc \in LOC_l \text{ and } sto \in STORE_l \\ cont(loc, sto) & \uparrow & \text{otherwise.} \end{array}$$

We assume that the evaluation function $exp\_ev : EXP \times ENV \times STORE \rightharpoonup VAL$ satisfies

$$\begin{array}{rcl} exp\_ev(\alpha, env, sto) & = & \alpha \\ exp\_ev(x, env, sto) & = & cont(address(x, env, sto_0), sto) \\ exp\_ev(a[exp_1, \ldots, exp_m], env, sto) & = & cont(address(a[exp_1, \ldots, exp_m], env, sto_0), sto). \end{array}$$

The address of a variable is computed by

$$address(x, env, sto) = cont(bind(x, env), sto),$$

while the address of an array element is computed (using *global* store $sto_0 \in STORE_g$) by

$$address(a[exp_1, \ldots, exp_m], env, sto) = base + (\ldots((i_1 dim_2 + i_2)dim_3 + i_3)\ldots)dim_m + i_m$$

where $bind(a, env) = \langle base, dim_1, \ldots, dim_m \rangle$, $i_j = exp\_ev(exp_j, env, sto)$ for $j = 1, \ldots, m$.

We also assume the following *coincidence property* for evaluation of expressions:

$exp\_ev(exp, env, sto) = exp\_ev(exp, env', sto')$ whenever, for all the variables and array elements ξ occurring in exp, we have $exp\_ev(\xi, env, sto) = exp\_ev(\xi, env', sto')$.

Memory Mapping

The full APE100 memory addressing model is obtained by extending locations to *addresses*:

$$ADDR = ADDR_g + ADDR_l, \quad LOC_g = ADDR_g, \quad LOC_l \subset ADDR_l.$$

While $ADDR_g$ is needed just to have a uniform notation, $ADDR_l$ contains the additional information required for the memory mapping feature. The APE100 local addressing model, which allows a node to access the local memory of a neighbouring node, is described by extending *cont* to addresses through:

$$cont(addr, sto_i) = cont(which\_loc(addr), sto_{neighb_{k(i)}}) \quad \text{for each } addr \in ADDR_l \setminus LOC_l$$

where $k = which\_neighb(addr)$. The partial functions

$$which\_loc : ADDR_l \rightharpoonup LOC_l \quad \text{and} \quad which\_neighb : ADDR_l \rightharpoonup \{1, \ldots, 6\}$$

yield the location and the relative position of the neighbour (i.e. left, right, up, down, back or front) respectively, while $neighb_k$ is one of the six functions

$$neighb_j : FPU \to FPU \qquad j = 1, \ldots, 6$$

which find the proper neighbour (according to the index j) of a given node, consistently with the topology of the machine.

Together with *cont* also the update write α to *addr* in sto_i gets extended to proper addresses, as

write α to $which\_loc(addr)$ in $sto_{neighb_k(i)}$

Logical conditions

Logical conditions are used in conjunction with **if**, **while** and **where** statements, and provide a way to control the execution of programs. Like all other terms in APESE, logical conditions can be either global or local.

The syntax of logical conditions is defined as usual, based on relational operators and logical connectives. In addition, quantifiers are introduced:

- if *cond* is a local logical condition, then **any** *cond* and **all** *cond* are global logical conditions.

We evaluate logical conditions by a function *cond_ev* in the expected way, using local (global) store for local conditions (global conditions not containing quantifiers). For **any** and **all**, we require:

- $cond\_ev(\textbf{any}\ cond, env, \vec{sto}) \Leftrightarrow \bigvee_{i=1}^{n} cond\_ev(cond, env, sto_i)$
  $cond\_ev(\textbf{all}\ cond, env, \vec{sto}) \Leftrightarrow \bigwedge_{i=1}^{n} cond\_ev(cond, env, sto_i)$

where $\vec{sto} = \langle sto_0, sto_1, \ldots, sto_n \rangle$, $sto_0 \in STORE_g$, $sto_i \in STORE_l$ for $i = 1, \ldots, n$.

To handle nested **where**'s, each $i \in FPU$ has its own "IF-stack" IF_i, where the current sequence of evaluated **where** conditions is held. If the conjunction of all the elements of IF_i evaluates to false, the FPU i is disabled, i.e. it performs no operation, except when evaluating a **where** condition, and pushing or popping its IF-stack.

Acknowledgement

We thank Raffaele Tripiccione, Angela Gelli and Gianmarco Todesco from INFN in Pisa and Rome for having introduced us into the secrets of APE100 and for their interest in our work.

REFERENCES

1. Y. Gurevich, *Logic and the challenge of computer science* in: E. Börger (Ed.), Trends in Theoretical Computer Science. Computer Science Press, Rockville MA 1988, pp. 1-57.
2. Y. Gurevich, *Evolving Algebras. A Tutorial Introduction*, in: Bulletin of the European Association for Theoretical Computer Science, no. 43, February 1991, pp. 264-284.
3. A. Bartoloni et al., *A Hardware Implementation of the APE100 Architecture*, in: International Journal of Modern Physics, C 4 (1993), p. 969.
4. A. Bartoloni et al., *The Software of the APE100 Processor*, in: International Journal of Modern Physics, C 4 (1993), p. 955.
5. The APE100 Collaboration, *"APESE" Language*, preprint A100/APESE/S-04, INFN, Pisa.