APE100 project

From APEWiki

This document describes a parallel processor for large scale scientific computations. This machine is based on the experience acquired with the APE computer. It is a fine grained SIMD processor, delivering a peak speed higher than 100 GFlop. This performance is, for instance, required for high precision computations in Quantum Chromo Dynamics [QCD]. The system and hardware architectures are described.

Contents

APE100 Architecture

APE100 is a SIMD (Single Instruction Multiple Data) parallel computer with a tridimensional architecture based on a cubic mesh of nodes. Each processing node is connected to its six nearest neighboursin all three space dimensions. Periodic boundary conditions are used to wrap around the mesh. The main advantage of this simple structure is modularity. The configuration of APE100 is scalable from 8 to 2048 nodes.

Each node is composed of a single precision 32 bits IEEE-754 floating point processor, (the MAD), a 4 (or 16) Mbytes RAM bank and the logic for node-to-node communication. Parallel processing on APE is limited to operations on floating point numbers. Each node is capable of 50 Mflop so that the complete configuration has a performance of 100Gflop. There is a Controller for each group of 128 nodes and all the Controllers execute exactly the same code.

The Controllers are based on a scalar processor named Z-CPU which executes the user program, all integer operations and delivers the microcode and the data addressing to the nodes. The Z-CPU has its own data memory, program memory and registers.

The APE100 synchronous core (i.e. the Z-CPU's and the nodes described above) is connected to a host computer through an asynchronous interface, called AI (Asynchronous Interface), based on a network of transputers which act as front end. The user program runs on the synchronous core.

This asynchronous interface is able to access all the resources of APE100: control registers and memory banks of the synchronous core as well as the parallel mass storage system. The host communicates with the AI by using transputer links. The system code running on the AI is able to convert requests coming from the host (on the transputer links) into operations on the APE100 machine, as well as to serve interrupts arising from the synchronous machine. Most part of the system software runs on the host computer while the real time kernel of the operating system runs on the AI.

The *Tube*

The *Tube* is the smallest APE100 unit. It contains 128 processing nodes, organised as a long and thin 2 x 2 x 32 mesh, as well as control and interface hardware. The main building block of the *Tube* are the Controller, the floating point unit (FPU) boards, each containing 8 nodes, and the backplane providing point-to-point communications and global control distribution.

Controller

There is one Controller on each *Tube*, responsible for the overall co-ordination of the system. In a SIMD processor there is a large amount of control circuitry that can be shared by all processing nodes.All such circuitry has been concentrated in the Controller. The board housing the Controller also contains the asynchronous circuitry needed for the interface to the host computer.

The main tasks performed by the Controller are program flow control, address generation and memory control. The first two task are potential bottlenecks for the performance of the array of nodes, if they are not fast enough to keep the nodes busy most of the time. Memory control for dynamic memories, on the other hand, is a complex activity, requiring sophisticated circuitry. This circuitry is shared by all nodes and is housed in the controller.

The main building blocks of this controller are a register file, an integer ALU and multiplier, and a three input adder used for address computation. At each clock cycle, the Controller is able to performe an integer operation in parallel with an address computation and memory access. Global program flow is controlled with standard conditional branch instructions. A parallel conditional branch, controlled by the logical AND (if all) or the logical OR (if any) of individual condition codes, evaluated on each node, is also possible.

Independent memory banks and busses for program and data storage and access are provided, so instruction fetch and data moves do not interfere. The program memory is 128 KWords deep, while the data memory contains 64 KWords (32 bit wide). The program memory word is 128 bit wide, of which 32 bits form the Controller instruction and the remaining 96 provide the control word for the processing nodes.

The memory controller for the nodes, handles all memory activities, including a vector access mode, and refresh control.

The Controller runs at 12.5 MHz frequency of the processing nodes.

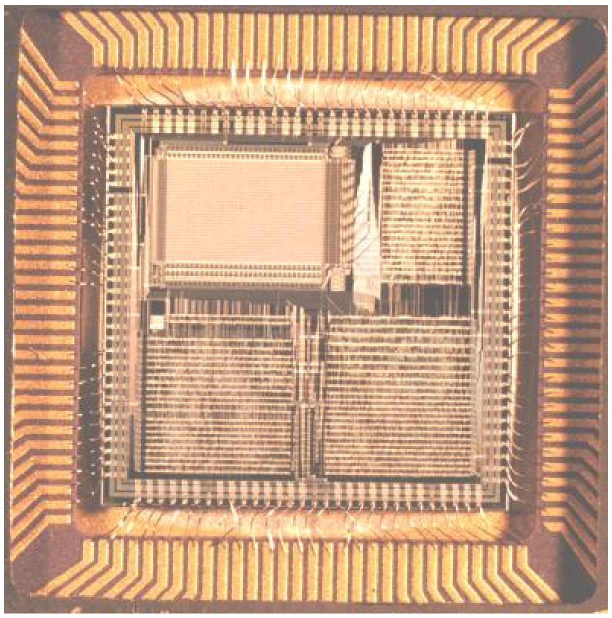
The processing node

The nodes of APE100 make up a 3D grid. A structure of 8 nodes, arranged as a 2 x 2 x 2 cube is the most convenient of such configurations, since it fully retains the symmetry of the larger machine. A processing board (PB) contains eight floating point units, their memory banks and the network interface circuitry.

The floating point nodes of APE100 are based on the MAD chip. MAD ia a custom VLSI pipelined device, driven by a 48 bit instruction word and running at 25 MHz. The Controller supplies instructions to the MAD chips at a rate of two every (12.5 MHz) clock cycle. The MAD chip contains a large register file (128 registers) and a floating point ALU and multiplier.

Several additional features complement this basic structure: a condition code stack, look-up circuitry for several mathematical functions, and an error detection and correction circuit. The decision to develop a custom device has permitted us to include in the chip all the needed support circuitry. In turn this made it possible to house eight processing nodes on the same physical board. The basic mathematical instruction is the so called "normal instructions":

```
A x (±B) ± C
```


MAD (Multiply and Adder Device)

Ordinary additions and multiplications are translated into the "normal". As an example *A x B* translate in *A x B + C*. Two of the MAD registers hold the values 0 and 1, needed for this translation. At each clock cycle the MAD can start a multiplication, a sum and one I/O operation. Therefore, the MAD chip is in principle able to perform one I/O operation every two floating point operations.

Memory

Node memories are implemented with dynamic 1M x 4 bit chips with an access time of 80 ns. Each memory bank is 1 MWord deep and each 32 bit word has 7 check bits to allow single error correction and double error detection. The bandwidth of the memory is 50 Mbyte/sec, i.e., one word every two MAD clock cycles. To reduce the error rate, the 4 bits belonging to each chip are assigned to different data words, so that multiple errors on a single chip will show up as single errors on different nodes.

Communications

All inter-node communications are handled by a communication controller and are transparent to the user. The access to neighboring nodes is memory mapped. In the high level language this translates into a very simple construct. If V(i) is the ith element of a vector in a given node, V(i + right) is the ith element in the right-neighbor node.

From the hardware point of view the neighboring node can either be on the same board or in a neighboring board, depending on the transfer direction and on the node being considered. This fact is however completely transparent to the user who, for programming purposes, has only to think of a regular array of processing nodes.

Achieving such degree of transparency is a complex task. Node memory accesses involve the cooperation of two main actors. The *Memory Controller*, housed in the Controller board, produces the control sequences for memory access, while the *Communication Controller* housed on each processing board, is responsible for inter-node data transmission. When the *Memory Controller* initiates a memory access, the *Communication Controllers* on each processing board decode the corresponding address. If a remote transfer is involved, the *Communication Controllers* remove the data words from the busses of all nodes and route them to their appropriate destinations. Such destinations are other node busses on the same board or the point-to-point links to the appropriate board on the backplane. The bandwidth for remote transfers is one fourth of the local memory bandwidth. The inter-node bandwidth is however large if compared with that available on MIMD machines, that also requirea large latency time synchronisation. The Communication Controller is based on yet another custom device, called the Commuter. Each Commuter handles a four bits wide slice of the data word. Ten Commuters are therefore required for each processing board. The Commuter is also used to interface node memories to the host.

The Backplane

The Controller and the processing boards share a communication backplane. The backplane provides bussed signals and power distribution, and point-to-point data links. Bussed signals provide instruction words for the nodes, addresses for the memory banks and a few control signals. Point to point links move data among logically neighboring nodes. As already explained, each board contains a 2 x 2 x 2 cube of processors. Adjacent boards continue the mesh in one of the three directions, to build up a closed torus of length 32. This torus can be split in two shorter (length 16) independent tori. Boards are arranged in the crate in such a way as to make all connections of approximately the same length. The backplane has about 80 point-to-point links stemming out of each board and about 140 bussed signals. Individual coaxial cables distribute the clock to the processing boards. This ensures synchronisation accurateto better than 1 nsec.

APE100 Software Survey

Preface

This paper describes the APE100 system software architecture and its major components: the compiler, the operating system and the debugger. The TAO language, as well as the debugger command language, are incremental languages based on Zz, an innovative dynamic parser. Zz and these language are shortly described in the following. Some major utilities, simulators and analysis tools are also described. [Back to section start]

Overview

We can divide the system software of APE100 into its major components: the compilation system, the operating system and the APE100 utilities. The symbolic debugger and some system services interact with the three components and require a separate discussion.

All the system software code is written in C-language to ensure portability. [Back to section start]

The compilation system

The compilation system runs on any conventional computer, usually the same host connected to APE100. Two types of source languages are supported by the compilation system: TAO and the APE100 assembler. Assembly language is used for writing diagnostic or system programs. The user is strongly suggested to write his programs using the TAO language: it is easier to use and produce object code which is more optimised.

The compilation system of APE-100 can be divided into two steps. The first one reads a source file and produces an intermediate APE relocatable code file (.IAR). This file can be linked with other modules in the second step. In the second step the linker collects the modules and the code generator produces a loadable and executable file (.ZEX).

In the case of an TAO source the first step of the compilation system uses the Zz dynamic parser linked with routines which perform some optimizations and produce the .IAR file and, optionally, a macro assembly source listing. The optimization done at this level is only oriented to reduce the number of memory to register I/O operations a much more sophisticated optimizer is being developed. The compiler, therefore, attempts to use as much as possible the machine registers. In this phase of the process floating point registers are considered as an unlimited resource. Floating point registers assigned in this phase are called virtual registers.

The .IAR format supports physical registers and virtual registers. The virtual registers are assigned to physical registers during the final optimization step. The APE assembly is in (near) one-to-one correspondence with the content of the .IAR file. The same routines used by Zz to produce the .IAR are called by a macro expander to compile macro Assembler source programs.

The main difference between a .IAR produced from the TAO language and from the macro Assembler is that the first uses virtual registers and relocatable addresses, while the second uses physical registers and memory locations (directly allocated by the programmer). The two kinds of optimization done from the second step are: Code compaction and temporized microcode generation.

One of the main tasks of the code compaction program is that of combining floating point additions and multiplications into multiply-and-add operations, which are called "normal" operations because the architecture of the floating point processor (MAD) is optimized for them. To increase the number of "normals" produced, the associativity of the add (and subtract) operations can be changed.

Other optimizations done by the current code reducer are:

- folding: compile-time execution of operations with known result, like multiplications by zero or by one,
- deletion of unnecessary move operations produced by the first step of the compilation chain,
- deletion of dead code (operations whose result is not used, like those arising from the trace of a matrix product).

The main task of the microcode generator is to rearrange the instructions in order to fill the pipeline. From the point of view of the microcode generator APE-100 can be treated like a machine consisting of two parts which are completely synchronous: the integer processor (the ZCPU), which is in charge of integer calculations, address generation and flow control, and the floating point processors (the MAD's) which do most of the number crunching. The internal architecture of these two parts is quite similar allowing a single code generation phase.

The MAD and the ZCPU are VLIW (very large instruction word) pipelined chips. A single instruction will affect different devices, which correspond to different instruction bit fields, at different times on both control words (the MAD control word and the Z-CPU one). In order to fill the pipelines, the MAD- ZCPU microcode producer is free to completely reschedule both floating point and integer instructions, with two constraints: no device can be taken by two instructions at the same time, and no instruction can execute before its operands are ready.

After scheduling, the code producer maps all the MAD virtual registers to physical ones. This phase does not concern the ZCPU part of the code, since its registers are always treated as physical. Compilation may fail at this point if there are not enough physical registers to satisfy all requests. Given the large size of the MAD register file (128 registers), this is a rare event, usually caused by an large use of Register and Phys Reg variables (see the TAO language manual). We hope in the future to improve the situation by the automatic "spilling" of excessive registers into memory. The problem of register spilling is a very complex one and has not been tacked yet.

Our optimization chain has been proved to be highly effective for many different codes. The performance of an optimised QCD code is near of 80% of the machine peak performance.

All the informations describing each assembly code: number of operands, result times, device occupation times, code bitfields, etc. are defined in tables which are read at the startup of the microcode producer. It's possible to compile these tables from a source file written in a machine description language suited for a general APE architecture and have them read by the compiler at startup. This feature is used to develop new hardware architectures, it was used to define the MAD processor and is now evolving to define more complex architectures.

The predecessor of the single-chip Controller, the ZCPU, was called S-CPU. It had an architecture different from that of MAD. The code production for machines with a S-CPU controller had to be performed in two distinct phases: one for flow control instructions and addressing and one for MAD' instructions. Only the MAD microcode was suitable for pipeline optimization.
[Back to section start]

ZZ and the TAO language

Dynamic compilers are compilers able to follow the evolution of a grammar during source program compilation. Dynamic compilers and growing grammars allow the compilation of programs written in programming languages adapted to the problem at hand. Some of us have defined a theoretical framework for "evolving grammars" and "dynamic compilers" [4].

Dynamic compilers easily perform purely syntactic strong type checking and operator overloading. A dynamic parser ("Zz Parser") has been developed and has been successfully employed by the APE100 group to develop the APE100 compiler, the symbolic debugger and other system software tools. Using Zz we have defined a starting language inspired by the familiar FORTRAN syntax. This starting language, TAO, is very similar to a new generation FORTRAN language with parallel support instructions, flow control instructions and integer, float and complex data types and operators. The programmers may use the Zz dynamical compiling features to extend the predefined TAO language, adding new syntaxes and features suitable for their applications and creating Language Extension Libraries (LEL). The starting language itself is defined in a LEL. We have built a number of LEL's. The QCD extension library ("qcd.hzt"), for example, defines the compilation rules for QCD specific data structures (e.g. su3 matrices and spinor data types) and makes available an appropriate set of QCD specific statements and operators (e.g. gamma operators, multiplications of su3 "variables", ...). Other problem specific LEL's have been defined for fluid dynamics, neural networks and for other promising areas of application of the APE100 machine.

The application code written by the APE100 users using an evolving language are several times shorter than programs written in conventional programming languages, due to the easy introduction of convenient operators, statements and data types. The opinion of the APE100 user community is that these factors greatly enhance the quality and readability of the resulting application programs.

The FORTRAN-like flavour of the TAO starting language will probably look familiar to any programmer of scientific applications. It is the smoothest transition path from a standard computing environment to the APE100 parallel environment. The TAO language is not a FORTRAN with a number of non portable extensions. Our programming language has been designed to allow easy and efficient coding of scientific applications on a parallel computer with unusual architectures. The actual implementation can be considered as a subset of a more general TAO dynamic language which has been defined by the APE100 collaboration. A typical novice user may well start writing programs in the TAO language. As he acquires more confidence in the language, he will probably include more and more advanced Zz features, in order to write more compact, readable and easily maintained programs.

Differences with more traditional FORTRAN-like languages can be noticed in two main areas: a) parallel features have been introduced in the language. As an example, parallel data structures can be defined, and new instructions are defined for (parallel) branch control. Several features of the language in this area have been tailored to the APE100 architecture. b) features typical of more structured object oriented languages (like the C++ language) have been introduced. An important example in this line is the definition of several data structures (e.g. SU(3) matrices and spinors) frequently found in QCD simulation programs as well as corresponding operators.

In the following we describe some of peculiarities which are strongly related to the hardware architecture of the APE100 computer family. Every time the APE100 programmer declares a data structure, an appropriate amount of local storage will be reserved for it on each node (at the same local address for all nodes) and logically associated to the name of that data structure.

All nodes of APE100 execute the same code (although acting on different data). This means that each operation written in TAO language, acting on some floating point variables actually activates the same operation on every node of the mesh. The data items contained in the local storage corresponding on each node to the program variables are the operands of the operation executed by that node. Integer variables are defined on the ZCPU only. Integer arithmetics in a typical APE100 programs should involve mainly (but not necessarily only) the evaluation of addresses for the local (floating point) variables, as well as some arithmetic or logic manipulation of control variables (e.g., loop indexes). The 32 bit integer data type should be used when declaring the addressing, loop indexing and array indexing variables stored on the "flow control and addressing unit" (ZCPU, Controller).

On APE100 there are three types of conditional branch instructions, according to the type of conditions which can be tested to alter the sequential flow of program execution. Two such instructions may cause an effective branch in the program flow. The third instruction is locally managed by each node and may result in a temporary suspension of the program execution on that node. The last feature allows a crude form of local program conditioning. The first type of condition is tested on integer variables defined on the ZCPU. This type of condition can be inserted in standard "if" statement. A second type of condition involves local data defined on each node of the array, but used in a global way. The control condition can be the obtain as the logical AND or the logical OR of the conditions evaluated on all nodes. This type of condition can be also inserted in "if" statements, alone or combined with the standard conditions discussed before.

```
Examples: {?}

        if All (local condition)

                { statements }

        if (Any (local condition) && (integer condition) )

                { statements }
```

In the first example the global condition is the AND of local ones. In the second a global condition results from the OR of local ones is combined (&& stands for logical AND)

The third type of condition is local to each processing node. This type of condition is used in a special TAO instruction, called "where". The "where" instruction does not alter program flow. Rather, instructions contained within the where clause are executed on only those nodes where the relevant condition is true. Instructions are converted to nops (no-operations) on those nodes where the condition is not met.

The APE100 architecture supports data access of each local node to its own memory bank, or to the memory banks belonging to its six nearest neighbours of the three-dimensional grid of processors. Access to memory banks belonging to far-away processors can of course be programmed as a sequence of hops among adjacent nodes. The memory bank selected for (local or remote) access is specified at run time by the program in terms of the highest significant bits of the address. An example will make the previous statement clearer. Assume that v[i] is an TAO vector. The program sequence a = v[2+RIGHT] copies onto variable a in each node the value v[2] taken fro the node on the right. RIGHT is an integer constant defined with the proper bit pattern. This operation is accomplished independently on all processing nodes. [Back to section start]

The operating system

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | **sales@docketalarm.com** | **1-866-77-FASTCASE**

fastcase® Smarter legal research.