Unconventional superspeed computer systems

by TIEN CHI CHEN

IBM San Jose Research Laboratory
San Jose, California

INTRODUCTION

As machine systems grow in complexity to provide ever higher quality and quantity of service, increasing attention has to be paid to aspects of computing which are not the requirements for problem solving, but rather demands imposed by the handling technique prescribed for earlier machines!

A study of the self-optimizing capabilities of large computing systems suggests that unconventional systems can be conceived which can gain efficiency, adapt to human usage patterns, and be consistent with future hardware promises.

SUPERMACHINE SYSTEMS

When hardware (and to an extent software) investment for a given system exceeds a certain critical size, facilities can be provided for self-optimization. The system no longer needs to process the workload passively, exactly as externally prescribed; it becomes a complex of autonomous units, each actively manipulating resources under its own jurisdiction. We shall designate such a large, self-optimized complex a "supermachine system." Self-optimization will be shown to be a practical necessity for throughput efficiency; conventional processing, by the same token, will be seen as a handicap.

PARALLELISM, PIPELINING AND EFFICIENCY

An important reason for large systems is throughput enhancement. Within a given technology, there are two basic techniques to achieve this end: (a) quantitative split of workload, that is, parallelism by identical processors, and (b) qualitative division of labor, an extreme case of which is pipelining.

Employing parallelism, a multiprocessor of multiplicity $M$ is made to sweep over job requirements to achieve complete coverage in space-time. The job to be processed has a time-dependent natural width function $w(t)$. For convenience of analysis we assume $w(t)$ to have only two values 1 and $W$, with time expenditures $t_1$ and $t_2$ respectively. Jobs with other $w(t)$ can usually be recast in terms of this standardized model (Figure 1).

The parallelism ratio $\rho$ can be defined by

$$\rho = \frac{(\text{space-time of the part of job with width } W)}{(\text{space-time of job})}$$

$$= \frac{W t_2}{(t_1 + W t_2)}$$

which is a property of the job. The multiprocessor needs to make only one pass to cover the narrow portion of the job and $n$ passes to cover the wide portion, $n = \text{the integer part of } [(w+M-1)/M]$. The efficiency of coverage is

$$\eta = \frac{(\text{space-time of job})}{(\text{space-time swept by multiprocessor})}$$

$$= \frac{1}{M} \left[1 - \left(1 - \frac{n}{W}\right)\rho\right].$$

as shown in Figure 2. If $W$ is not exactly divisible by $M$, there will be a residual inefficiency even when $\rho = 1$.

$\eta$ becomes $1/k$ when $1 - \rho = (kW/M - n)/(W - n)$. Figure 3 shows that, for $W = 32$ and $n = 1$, even if $M$ exactly matches $W$, $\eta$ falls to 50 percent, 20 percent and 10 percent while $\rho$ is 96.8 percent, 87.1 percent, and 71.0 percent respectively; and for $W = 100 = M$, $\eta$ = 10 percent when $\rho$ remains as high as 90.9 percent. Further, no matter how large $M$ is, the effective multiplicity $\eta M$ can never exceed $1/(1 - \rho)$.

Instead of many identical general processors, the division of labor approach employs a collection of non-identical, specialized mechanical operators functioning simultaneously to form a general facility. An extreme form of this is pipelining.

A simple pipeline is constructed by first analyzing a given computing process into a linear sequence of
Figure 1—Job profile for the inner loop of a $32 \times 32$ matrix multiplication

If $M = 31 = W - 1$, $\eta = 0.52$ when $p = 1.0$

Therefore, though details differ considerably, essentially the same efficiency worries plague the parallel and simply pipelined systems. Realistic superspeed systems often involve networks of unevenly quantized pipelines and nonuniform parallelism. The job stream may not possess ideal multiprocessing characteristics, and the efficiency becomes even more elusive. Self-optimization then becomes a critically needed feature, to allocate available resources to maximize throughput, yet preserving the correctness of results.
SELF-OPTIMIZATION

There are two main categories of optimization:

(a) Procedure redefinition
(b) Resequencing of the procedure collection.

Redefinition calls for much perception and a global comprehension, and tends to fall in the domain of *external optimization*, by human or compiler, although the IBM System/360, Model 90 series show a modest ability. ²

Resequencing of procedures is a chief ingredient of multiprogramming. ³ By handling many procedures together, the total equipment needed for adequate throughput will approximate the sum of the average (rather than the peak) requirements. Coincidental peak requirements often can be resolved, and under-usage of equipment avoided, by permuting the workload sequence.

To be truly effective, this freedom to deploy available resources should extend down to each autonomous unit, within the scope of which the input work stream(s) can be analyzed into a number of causally independent streamlets, behaving like requirements from different users. Available resources are then assigned to enhance average progress without completely ignoring any one streamlet. Thus, the supermachine system is expected to practice not only multiprogramming in the usual sense of the word, but also multiprogramming in the small, or “micro-multiprogramming” (Figure 6). The workload characteristic to be exploited is job independence, rather than the much more restricted job parallelism.

The control technique is generalized table management. Each unit will alter linkages, recode, reclassify, change priorities, resolve conflicts, handle errors and anticipate future events. The hardware requirement consists mainly of memories for waiting streamlets and encoded signals, and capabilities for selection and switching; in other words associative memory co-packaged with standard logical hardware. Such a distributed intelligence is costly with current technology, but can exploit LSI for its speed, economy, compactness, also its penchant for regularity.

CONVENTIONAL PROCESSING

To achieve effective self-optimization, the system should be granted maximum freedom to redefine, or resequence the work within its power. This implies *minimum over-specification* by external agents. The real-time happenings, because of multiprogramming, overlapped I/O, interruption features and conditional branches, are virtually impossible to anticipate. Detailed specifications, made without the intimate knowledge of real-time events, tend to handicap the self-optimization.

Most problem programs today are written in procedural languages. These programs are pre-processed by a compiler into machine instructions, to be subsequently handled explicitly by the machine. The compiled machine codes historically were the only messages...
PROCESSING MULTI-OPERATOR STATEMENTS

Consider the following FORTRAN innerloop, important in solving simultaneous equations:

\[
\text{DO 300 } \quad K = J, 100 \\
300 \quad A(I, K) = A(I, K) \times Q - A(J, K)
\]

During compiling, statement 300 is recast into instructions. For one- and two-address schemes, the arithmetic part reads as follows.

\[
\begin{align*}
C(A_{ak}) & \rightarrow C(R0) \\
C(R0) \times C(Q) & \rightarrow C(R1) \\
C(R1) - C(A_{ak}) & \rightarrow C(R2) \\
C(R2) & \rightarrow C(A_{ak})
\end{align*}
\]

Pipelined arithmetic operators \( M, A \) can be installed for multiplication and addition, each with a throughput of one result per cycle. Therefore, one could rightfully expect an overall throughput of one matrix element \( (A_{ak}) \) per cycle. However, the instruction processing stream crosses upon itself three times at \( R0 \), thus slowing the flow to \( \frac{1}{6} \) the expected maximum rate (Figure 7a).

The use of a three-address code

\[
\begin{align*}
C(A_{ak}) & \rightarrow C(R0) \\
C(R0) \times C(Q) & \rightarrow C(R1) \\
C(R1) - C(A_{ak}) & \rightarrow C(R2) \\
C(R2) & \rightarrow C(A_{ak})
\end{align*}
\]

results in the diagram in Figure 7b. There is no pipeline crossing, and unit throughput can be maintained. However, the redundant use of the three registers is apparent, and the pipeline is lengthened unnecessarily; also three-address codes usually consume more program space.

A still better way is to apply surgery on the instruction sequence to eliminate the redundant paths, resulting in the clean pipeline in Figure 7c.

\[
\begin{align*}
C(A_{ak}) & \rightarrow C(R0) \\
C(R0) \times C(Q) & \rightarrow C(R0) \\
C(R0) - C(A_{ak}) & \rightarrow C(R0) \\
C(R0) & \rightarrow C(A_{ak})
\end{align*}
\]

This technique, employed in real time, is found in the S/360 Model 90 series of computers.\textsuperscript{2,3} It re-creates the original string of operations. Clearly, hardware would be saved and compiling simplified if the fragmentation into instructions never had taken place.

ARRAY PROCESSING

The program in the last section involves the systematic handling of vectors. In routine computing involving arrays, the system is only given piecemeal information about the individual array elements, making it hard to divine the intent of array processing. Consequently, undue burden is placed on the decoding mechanism and the memory access hardware, and arithmetic pipelines tend to run at fraction capacity. There is the extra cost to perform index arithmetic to locate the array elements one at a time, and execution of branch instructions to ‘close the loop.’

Concise descriptions of array procedures already exist in APL/360\textsuperscript{5} and to a lesser extent in PL/1, and are easy to use. Should these specifications remain unaltered by the act of compiling, the supermachine can be designed to mobilize its resources in real time.
Memory data can be accessed en masse. Arithmetic operators can be reserved, and pre-configured into appropriate pipelines. Work areas can be created. I/O devices can be synchronized. Competing programs can be downgraded in priority. In general, near-peak efficiency can be achieved for the self-optimized supermachine system.

NAME HANDLING

During compiling, the names in the procedural language program are mapped into addresses.

Various relocation schemes further map the addresses into some other addresses. "Paging" and "cache" mechanisms perform dynamic mappings in real time, based on information not available to the compiler, and have tremendous performance implications to large machines. The initial mapping by the compiler, not capable of replacing paging and caching, is at least partly redundant.

Most procedural languages today use names; the latter's value to the human user lies not just in the mnemonic value, but in the freedom to associate. A name is, so to speak, a universal reference-quantum. Names are used to designate words, arrays, bits, character strings, and structures. They also designate branch targets, subprograms, and aspects of the operating system. They may designate other names, emptiness (null) or a state of incomplete definition. One entity may have several names, or several items may have the same name, the ambiguity being resolvable by context.

Direct handling of names by the machine not only removes a redundancy; its implication to the human user will be profound. During debugging, the correlation between the machine code and human procedural program will be more easily understood, and there is the possibility of following and manipulating associations efficiently during computation.

The handling of array names will permit the real-time assignment of space to fit the needs exactly. The programmer need not specify array dimensions, and indeed can alter them during computation. Already this freedom is being exploited to great advantage in APL/360 and PL/I. The rigid dimensioning of arrays is proving to be a fetter for both man and machine.

MACHINE "M" FOR MULTI-OPERATOR STATEMENTS

The unconventional supermachine systems are indicated in Figure 8. One can consider first Machine M, which uses (variable length) multi-operator statements, possibly with rigid field formats and conventional addressing. Decoding will become more orderly, and the intermediate register assignment will be entirely up to the supermachine system in real time.

As each statement describes a causal chain of events, at any time normally only one member of the chain can be handled. The unfinished statement segment can be put in a waiting state; other statements bearing no causal interlock can be processed simultaneously. The hardware to detect and resolve logical conflicts, and to redefine procedures (such as that used for internal-forwarding) will be simplified.

MACHINE "A" FOR ARRAY PROCESSING

Array processing is one of the most-discussed unconventional machine features, but is usually presented outside the context of general purpose computing. Machine A is meant for array processing consistent with multiprogramming through self-optimization. It would use special array instructions, possibly created from a special compiler. There is, however, no need to prevent the system from handling its regular computing load.

When arrays A, B, interact to produce array C, elements within each array often are not causally linked, and thus can operate independently to fill an arithmetic pipeline. Array processing is then just a special form of multiprogramming, in which the 'jobs' are extremely well-defined, and are known in advance to be independent.

In Machine A, arithmetic for large arrays can be assigned a high priority and behaves like, say, up to 16 programs in parallel. Other jobs are given lower priority to ensure non-conflict. The handling, however, must be such that all jobs are processed sooner or later. Since array processing does not tax the decoding mechanism
too heavily, it is actually desirable to mix-in conventional jobs which are decode-limited.

Both statement processing and array processing enhance performance and are mutually consistent. It is possible to consider them together, to yield Machine "MA." The format incompatibility can be resolved by adopting statements to array processing also.

We observe that procedural languages like APL/360 and PL/1 already have array facilities, and can be used as a basis. These languages also have a distinct name-orientation which seems particularly advantageous for arrays.

NAME HANDLING MACHINES

To handle names directly in all ramifications, would seem to require significant overhead beyond those needed for multi-operator statements, array processing and standard self-optimization. This is because associated with each name is information concerning attributes and whereabouts of the named object. Although the handling technique is again table management, consistent with self-optimization needs, nevertheless, the resources needed will not be trivial, and except for special purpose tasks (such as text handling, list processing), name handling should combine with statement processing and array handling.

MACHINES "AN" AND "MAN" WITH ARRAY NAMING

Machine AN handles only those names affecting arrays, and is organized like Machine A. MAN is similarly Machine MA with array naming.

The simplest way to achieve array naming is to compile array names into indirect addresses, and a dynamic array management system is triggered on each referral. This is reminiscent of interpretive matrix arithmetic software which dynamically manages a shared memory pool. During computation both the array characterization and array contents can change.

As arrays increase in size, the name-handling overhead rapidly becomes insignificant relative to compute cost. It is interesting that, at the same time, dynamic memory management is truly meaningful with array naming, and is most beneficial when storage space is at a premium, namely when arrays are large. The hardware realization of AN and MAN will further allow much smaller arrays to be efficiently treated under extensive multiprogramming.

The concatenation of vectors to form a longer vector will be automatically accomplished; when these vectors are character strings, this facility will have a clear impact on the automatic handling of procedure language statements and textual material.

MACHINE "MAN"

The above technique can be extended to all types of namable information, but the overhead cost would be relatively large if the associated data processing time is small. Machine MAN attempts to pay an overhead, in order to reduce the overhead due to conventional compiling. It is a fully interpretive machine.

Here a procedure language is executed more or less directly. The system will have powerful facilities both for scanning for delimiters and for I/O conversion.

The choice of procedural language will have a strong bearing on machine efficiency. The APL/360 language seems closest to the ideal, because of its conciseness, array orientation, excellent non-numeric handling capabilities with which most other procedural languages can be simulated. Evén the strict (right to left) execution sequence is an asset for hardware implementation.

Once the procedural language has been decided upon, an important machine-suitable canonical subset can be selected for hardware implementation. Canonical statements will be handled efficiently. More complex statements can be treated by slower means, which definitely includes software assistance.

One can, therefore, write a code only the innerloop of which is written in canonical form, gaining both efficiency and convenience. Or, the user can submit a non-canonical code, and during execution a canonical version can be created and used. This is analogous to the "loop mode" feature in the System/360 Model 90 series, where the first traversal of the loop lays the groundwork for the subsequent efficient computation. The distinction between compiling and interpretation tends to disappear.

THE HUMANIZATION OF THE COMPUTER

The concept of computer instructions is a quarter-century old, and compilers have been in use for more than a decade. Both have a genuine reason for their invention, and are still playing a vital role in computing today.

Their introduction represented giant strides toward humanizing the computer, but not necessarily the ultimate step.

Within the computer complex, it has become known that the most valuable resource is the human user. Into the global cost-effectiveness equation must be factored his throughput, turnaround time, endurance limit and ability to learn. These are tied in with his
physiological makeup and behavioral patterns which have withstand the test of millennia.

Circuitry and memory costs represent but a small function of the total cost of a computer installation. The projected sharp drop due to large scale integration, when applied to conventional designs, may mean a small lowering of the user's total cost for the same performance level. By the same token, the doubling or even quadrupling of circuitry and memory would represent a minor increase of total cost. Yet the added hardware, if placed conventionally, will tend not to have too significant an effect on systems whose throughput bottlenecks lie in the costly I/O and auxiliary memory equipment.

Unconventional designs, on the other hand, can enhance both internal supermachine efficiency and human effectiveness. Each design represents an aspect of "language-directed computer design" as seen from a self-optimizing supermachine viewpoint, consistent with the advance of future hardware, and made possible by the recent advances in the codification of software techniques. The future compiler or prepossessor, freed of the drudgery of detailed tactical machine code specifications, can now emphasize the strategic deployment of resources towards global optimization of throughput performance.

REFERENCES

1 T C CHEN
Parallelism, pipelining, and computer efficiency
Computer Design Vol 10 pp 69-74 1971

2 T C CHEN
The overlap design of the IBM System/360 Model 92 central processing unit

3 D W ANDERSON et al
Machine philosophy and instruction handling

R M TOMASULO
An efficient algorithm for exploiting multiple arithmetic units
Ibid pp 25-33

4 E F CODD
Multiprogramming
Advances in Computers Volume 3 pp 78-155 1962

5 A D FALKOFF K E IVERSON
The APL/360 terminal system
Interactive systems for experimental applied mathematics

6 B RANDALL C J KUEHNER
Dynamic storage allocation systems
Comm ACM Volume 11 pp 297-305 1968

7 J S LIPTAY
Structural aspects of the System/360 Model 85. II: the Cache
IBM Systems J Volume 7 pp 15-21 1968

C J CONTI
Concepts for buffer storage

8 F H BRANIN et al
An interpretive program for matrix arithmetic
IBM Systems J Volume 4 pp 2-24 1965

9 M G SMITH W A NOTZ
Large-scale integration from the user's point of view
Proc AFIPS Fall Joint Computer Conference 1967 pp 87-94

10 W M McKEEMAN
Language-directed computer design
Proc AFIPS Fall Joint Computer Conference 1967 pp 413-417

C McFARLAND
A language-oriented computer design
Proc AFIPS Fall Joint Computer Conference 1970 pp 629-640