Automatic tuning of computer architectures

by KEN SAKAMURA, TATSUSHI MOROKUMA and HIDEO AISO

Keio University
Yokohama, Japan

and

HAJIME IIZUKA

Electrotechnical Laboratory
Tokyo, Japan

INTRODUCTION

One of the most important unsolved problems in the design of a computer system is the automatic optimization or tuning of the computer architecture to better suit the problem under consideration. In particular, it is very important to make an effective mapping of the structure of the problem to be solved to the structure of the computer being used.

The best approach to this problem is to automate the tuning procedure of the computer architecture by the computer itself, in which the computer evaluates its performance and thereby achieves better performance. This seems to be the best approach because the existing computer systems have become too large and too complicated to be handled by manual tuning techniques. It should be noted that tuning iteration is carried out automatically at the Instruction Set Processor (ISP) level utilizing the inherent characteristics of dynamic microprogramming.

The purpose of this paper is to present a detailed algorithm for an automatic tuning of computer architectures using dynamic microprogramming techniques and to show the effectiveness of the proposed ideas by describing the results of some experiments. We have been doing research on this problem since the late 1960s. Very little research has been conducted with regard to the automatic tuning of computer architectures.

Monitoring of programs

Detailed information regarding the dynamic characteristics of both the computer and the program to be solved is necessary to perform the optimization processes. This information must include the relative frequencies of machine instructions, the relative frequencies of sequences of instructions, that is serial dependencies, and the relative frequencies of address and data values. A monitor implemented in hardware, software or firmware collects this information.

Analysis of data

The information obtained from the monitor is processed by an analyzer which may be an independent computer. The function of the analyzer is to identify all possible candidates for instruction patterns that can be tuned up in order to create new instructions to save execution time and storage space. This is performed by a thorough analysis of the application program and its execution profile.

Feedback to computer

New instructions, that were synthesized by the analyzer, are converted into new microprograms which occupy less storage space and have faster execution time than the original instructions. The synthesized microprograms are loaded into the writable control storage (WCS) in the computer through the feedback path during the program execution to form a new enhanced architecture. This results in dynamic modification of the computer architecture and in better performance. It should be noticed that dynamic microprogramming techniques are very effective to realize this dynamic modification.

PRINCIPLES AND IMPLEMENTATION MECHANISMS

Basic principles

A simplified model of a mechanism for automatic tuning of a computer architecture at the ISP level is shown in Figure 1. The basic functions required for the optimization of the architecture or tuning are as follows:
Learning of tuning behavior

A special feature of our method is the fact that a data base is provided in the analyzer. The above-mentioned tuning processes are usually repeated until the desired performance improvement has been achieved. Therefore, the analyzer keeps track of the tuning behavior in the data base whenever tuning occurs. The analyzer can refer to the contents of the data base so that the number of tuning iterations will be minimized.

Definition of terms

The terms used in the next two sections are defined as follows:

Machine

Each intermediate language (IML) instruction set defines a corresponding machine, $M_i$, which has an IML instruction set $IML_i$. IML is a set of instructions represented as follows:

$$IML_i = \{I_1, I_2, \ldots, I_k, \ldots, I_n\}$$

where $I_k$ is an instruction which is a microroutine consisting of a sequence of microinstructions.

Program

A program is a sequence of IML instructions.

Block

In order to define a block, IML instructions have to be divided into the following two types:

1. Branch type—Instructions whose next instruction to be executed is not necessarily next one in sequence.
2. Sequential type—Instructions whose next instruction to be executed is always the next one in sequence.

A block is a static sequence of instructions which satisfies the following conditions:

1. Its entry point is restricted only to the top instruction.
2. The last instruction is of branch type.

Length of block

The length $\gamma(B)$ of a block $B$ is the number of instructions contained in that block.

Instruction pattern

An instruction pattern $P$ is a continuous sequence of instructions in a block. The number of different instruction patterns in a block is not greater than $\frac{1}{2}\gamma(B) \cdot (\gamma(B) - 1)$.

Length of pattern

Let the bit length of pattern $p$ consisting of $\gamma$ be $\gamma(p)$.

Weight of a pattern

The weight of a pattern $p$, $\xi(p)$ is defined as follows:

$$\xi(p) = \frac{\sum T(p) \times f(p, i)}{T_0 \times \gamma(p)}$$

where

- $T(p)$: Execution time of pattern $p$
- $f(p, i)$: Number of times that pattern $p$ is executed in the block $i$.
- $f(p)$: Number of times that pattern $p$ is executed in all blocks.
- $T_0$: Total program execution time.
- $\gamma(p)$: Length of pattern $p$.

Improvement ratio

The capacity of writable control storage required to convert a pattern $p$ into a microprogram may be thought of as the overhead of the tuning. It is represented by $\theta(p)$. Fur-
thermore, the improvement ratio \( (\mu(p)) \), indicating how much improvement of the execution speed has been achieved by tuning, is defined by the following equation:

\[
\mu(p) = \frac{T(p) - t(p)}{T_0} = \frac{\tau(p) \times f(p)}{T_0} - \frac{\tau'(p) \times f(p)}{T_0} = \frac{\tau(p) - \tau'(p)}{T_0} f(p)
\]

where

- \( T(p) \): Total execution time of pattern \( p \)
- \( t(p) \): Total execution time of pattern \( p \) after tuning
- \( \tau(p) \): One execution time of pattern \( p \)
- \( \tau'(p) \): One execution time of pattern \( p \) after tuning
- \( T_0 \): Total program execution time
- \( f(p) \): Execution frequency of pattern \( p \) in all blocks

**Optimum design of an intermediate language based on dynamic computer behavior**

It is well known that better execution efficiency may be achieved through description of the problem in a lower-level language. For instance, better performance can be expected when the problem is written in machine instructions rather than in high-level languages, and much better performance can be expected when the problem is written in microinstructions rather than in machine instructions.

The direct description of a problem in microinstructions is called “Firmware,” and this results in the creation of new instructions suitable for the problem to be solved. The improvement of the execution efficiency is carried out at the ISP level, and this procedure is called “Architecture Tuning.”

To achieve optimal efficiency, it is clearly desirable to implement all the programs in firmware. However, more than 80 percent of the execution weight is concentrated on at most four to five percent of the total number of program steps. Therefore, the best performance cost ratio is not achieved by making all the programs into firmware. Parts of the program should be implemented in firmware considering the dynamic behavior of the program execution. In this case, the best strategy is to apply microprogramming to those parts of the program, which are most frequently executed. In order to automatically detect the parts to be tuned up, the program is divided into blocks based on the proposed algorithm described in the next section. Next, the execution weight is measured to decide the blocks that have to be implemented in firmware starting from the block with the highest execution weight.

**An algorithm for tuning of IML instruction patterns**

An important consideration is that detailed information about the characteristics of programs is necessary to perform the tuning processes. Since the use of each IML instruction is not always uniform at the moment of program execution, the relative frequencies of both machine instructions and sequences of instructions, that is, serial dependencies, must be included in this information. The sequences of pairs or multiplets of instructions, that create new instructions are called “Instruction Patterns.” Thus, the programs can be expressed in terms of a finite number of instruction patterns by static analysis described below.

**Weighting of the instruction patterns**

In order to give a weight to each of the instruction patterns, the frequencies of sequences of instructions should be measured. In this case, no branch instruction should be contained in the instruction patterns. The program is divided into blocks separated from each other by branch instructions. The frequencies of the blocks are measured by a hardware or firmware monitor. The weight of the corresponding instruction patterns can be calculated from the measured frequencies of the blocks.

**Selection of the instruction patterns to be tuned up**

Many strategies have been considered to select the instruction patterns to be tuned up. We propose the following two methods:

1. Method to achieve maximum execution efficiency.

   Using the equation for estimating the tuning effect, which will be described in the next section, we can estimate the performance improvement \( (\mu) \) by means of the previously found instruction patterns.

   First, we select the instruction patterns with the maximum value of \( \mu \). It turns out that the weight of the other instruction patterns may be changed, since some of the instruction patterns may be overlapping each other and some instruction patterns contain other instruction patterns.

   Next, we carry out the static analysis again to weigh the instruction patterns. The value of \( \mu \) is recomputed to select new instruction patterns. The above steps are repeated until the sum of all the \( \mu \)'s exceeds a certain percentage which has been determined through some experiments.

2. Method to achieve maximum economical effect.

   This method is similar to the method described above, but now we use \( \mu/\theta \) instead of \( \mu \). Here, \( \theta \) represents the estimated value of the capacity needed for the WCS, which is necessary for the translation into firmware of the instruction patterns. It is also derived from the equations for estimating the tuning effect. Tuning is executed until the following condition is satisfied:

\[
\mu > \delta \text{ or } \theta > \delta'.
\]
Synthesis of new instructions

We make a selected instruction pattern into a new instruction. Each sequence of instructions is implemented in a single microprogram, which occupies less space and has a faster execution time than the original instructions. The tuning is based on the existing microprogram optimization techniques which bring about reduction in the number of memory references, efficient use of internal resources and the possibility of parallel processing.

Feedback

The synthesized instructions are stored into the WCS and also registered in the code generation part of the compiler. The codes being compiled are translated into corresponding new codes by applying an editing operation at the machine code level.

Estimation of tuning effects

If no limitation is imposed on the capacity of the WCS, then optimal tuning may be achieved by synthesizing new instructions corresponding to the instruction patterns detected during the program execution. However, the capacity of the WCS is directly related to the cost of the system. It is also necessary to save time and effort in writing the microprograms.

It is difficult to give a quantitative evaluation of this effort. However, if we assume that the amount of work spent on microprogramming is proportional to the number of the microprogrammed steps, then the amount of labor will be proportional to the capacity of the WCS. Therefore, it is desirable to select instruction patterns which may combine maximal efficiency with minimal capacity of the WCS. To accomplish this, a simple method to estimate the firmware effect, that is the ratio of the execution efficiency improvement to the increase of the capacity of the WCS, when instruction patterns are implemented in firmware, must be developed. If such an estimation is possible, the selection of instruction patterns can be done based on the overhead of the tuning.

Here, we assume that the efficiency improvement resulting from the implementation of a microprogrammed new instruction (μ) is a function of the weight of the instruction pattern implemented in firmware (ξ) and the increase of the capacity of the WCS (θ), that is,

\[ \mu = f(\xi, \theta). \]  

Further, we assume that θ is a function of the length of the instruction pattern (γ), i.e.

\[ \theta = g(\gamma). \]  

We determined the function \( f \) and \( g \), experimentally.

In order to determine \( f \), we determined \( \log(\mu/\theta^m) \) as a function of \( \log \xi \). The result is shown in Figure 2, indicating an approximately linear relation. From Figure 2, we obtained

\[ \log(\mu/\theta^m) = n \log \xi + a, \]  

hence;

\[ \mu = A \theta^m \xi^n, \]  

where \( a, A, m \) and \( n \) are constants determined by the IML instruction set under consideration.

Similarly, we obtained the relationship between \( \theta \) and \( \gamma \), which is indicated in Figure 3. The relation is given by a linear expression:

\[ \theta = b \gamma + c, \]  

where \( b \) and \( c \) are constants determined by the IML instruction set.

Equation 4 includes that firmware effect can be estimated from the weight of an instruction pattern and the total steps.
Automatic Tuning of Computer Architectures

Maximum efficiency improvement

What strategy do we choose? Maximum efficiency improvement or Maximum economical effect.

Select pattern with the maximum value of \( \dot{D} \).

Select patterns in data base as candidates, then select the pattern with the maximum value of \( \dot{D}/\dot{S} \).

Once more perform static analysis of the program to check the change in execution frequencies of the patterns, provided that new instructions are synthesized from the selected patterns.

Recalculate the weight based on the execution frequencies obtained above.

Calculate \( \dot{D} \) for all the effective patterns.

Does the total sum of \( \dot{D} \) for the candidates of patterns exceed the value of \( \dot{S} \)?

Yes

No

Does \( \dot{D} \) for the candidates exceed the value of \( \dot{S} \)?

Yes

No

Implement the selected patterns which are not registered in the tuning data base, in firmware.

Recompile the program after the selected patterns are defined as new IML instructions.

Measure the efficiency improvement by means of the new IML instructions.

Register the patterns corresponding to the new IML instructions in the tuning data base.

Update the tuning curve based on the information obtained above.

End

Divide program into blocks at IML instruction level. Assign a number to each block.

\( B(1), B(2), \ldots, B(n) \)

Extract possible instruction patterns from each block. Assign a number to each pattern: \( P(p) \), where \( p \) is pattern number. If \( P(p) = P(p') \), then \( p = p' \).

Measure execution frequencies of each block \( f(B) \).

For each \( P(p) \), calculate execution frequency \( f(B) \).

For each \( P(p) \), calculate weight \( (l(p)) \).

Select candidates for instruction patterns to be implemented in firmware as described below.

Calculate \( \dot{D} \) from Equation (4).

Is each of the candidates registered in the tuning data base?

Yes

No

Calculate \( \dot{D} \) for the candidates from \( f \) and \( t' \) obtained from the tuning data base. Give priority to the candidates.

Is calculation of \( \dot{D} \) completed for all instruction patterns?

Yes

No

Figure 4—Simplified flowchart of the tuning algorithm.
required for the corresponding microprogram. Therefore, this relationship is called the "tuning curve."

Obviously, the constants $a$, $A$, $m$ and $n$ are positive and the larger the weight $\xi$ and the increase of capacity of the WCS $\theta$ become, the greater the efficiency improvement will be.

A simplified flowchart of the algorithm just discussed is shown in Figure 4.

**Implementation**

It should be noticed that the tuning procedure just described can be performed automatically. The following implementation mechanisms are required:

1. Either hardware or firmware monitor is used to determine the weight of the instruction blocks.
2. A dynamic microprogramming technique is employed to modify the contents of the WCS.
3. The compilers for high-level languages should be able to expand the IML instruction codes. A technique used for incremental compilers can be adopted for this purpose.
4. The mechanism for the synthesis of new instructions can be implemented on a minicomputer or microcomputer.

**EXPERIMENTAL SYSTEMS**

We call the tuning mechanism consisting of a Monitor, an Analyzer and a Synthesizer the "Automatic Performance Evaluator (APE)." We developed two simple APE mechanisms to prove the effectiveness of the principles described in the previous chapters.

**Experimental System-1 on a HP-2100 and hardware monitors**

The process flow of tuning and learning in the experimental system is shown in Figure 5. We used a HP-2100 computer as the host computer to be tuned up. The HP-2100 computer is a 16-bit microprogrammable minicomputer. Since the machine instructions are incorporated in HP-2100, we can utilize two accumulators, while six other registers can be used with the micro-instructions.

The control storage consists of a ROM board with a capacity of 256 24-bit words in which the microprogram that has to interpret the machine instructions is stored, and three WCS boards for user microprograms. Each WCS board has the same as the ROM.

We employed a DYNAPROBE 7900+8000 hardware monitor of COMPRESS Co. as the APE monitor. We used...
a PDP-11V03 system to analyze or synthesize the output derived from the hardware monitor and further a D7916 count/time type hardware monitor and a D8028 map/store type hardware monitor to accumulate periods of time during which certain events have happened.

Figure 6 shows the block diagram of the experimental system. The program to be measured is executed on the HP-2100 host computer. The D7916 and D8028 hardware monitors collect signals generated from the host computer through probes in order to measure the weight of the instruction patterns.

The signals collected by the hardware monitors are directly supplied to the PDP-11V03 on which tuning analysis is carried out, the LSI bus of the PDP-11V03 is connected to the I/O bus of the HP-2100 through the I/O interface in order to make a feedback loop.

The LSI-11 is used for the analysis of instruction patterns, the reconfiguration of the IML instruction set and the organization of the data base in the tuning phase. To form a new IML instruction set, the microprograms for the synthesized instructions are stored into the WCS, that is incorporated in the HP-2100 through the I/O interface.

One of the main features of this system lies in the fact that there is no overhead in the monitoring function at all, since high-speed hardware monitors are used, and since the LSI-11 is exclusively used for tuning analysis. Figure 7 shows several equipments of the experimental system.

Experimental System-2 on a Burroughs B-1700

The Burroughs B-1726 is a microprogrammable computer with a data length of 24 bits. It provides many internal resources, such as four general registers and thirty two scratchpad registers. It can perform bit addressing and it has access to any sub-field of the registers. The capacity of the WCS which stores microprograms expressed by 16-bit words is 4K words.

We developed several S-code interpreters in which a firmware monitor was implemented. To measure frequencies of instruction patterns, a firmware monitor was incorporated in the microinstruction fetch routine in the S-code interpreter. Therefore, apart from the B-1700 computer no special hardware is necessary. Hereby, the monitoring overhead becomes larger and the time required for the instruction fetch is approximately twice as long as that in the original computer.

However, the tuning operation does not occur very often, and if it occurs, the microprogram for the monitor is removed and the S-code interpreter is reconstructed so that the total overhead does not become very large.
IMLs used for the experiments

For the IMls to be tuned up, we chose the following two languages:

1. The IML for PASCAL which incorporates 60 instructions and is emulated on both the HP-2100 and B-1700 computers.
2. The FORTRAN IML for the HP-2100, i.e. sequences of machine instructions interpreted on the HP-2100 computer.

RESULTS OF THE EXPERIMENTS

Tuning results

Some tuning results are shown in Figure 8—Figure 11. Figure 8 indicates the tuning results when a sorting program is executed on the PASCAL machine emulated by the HP-2100. For the tuning procedure on the B-1700 PASCAL machine, see the Appendix.

If the strategy selected is maximum execution efficiency, it turns out that the execution efficiency is twice as high as that of its corresponding non-tuned version, while the required capacity of the WCS is increased by 210 words when seven instruction patterns are selected. Further, the amount of the object code is reduced from 392 bytes to 284 bytes. This indicates that an effective code compaction is carried out during the static analysis of the program.

If the strategy selected is maximum economical effect, the capacity of the WCS is increased by 130 words, while the overall execution time of the program is reduced by 30 percent.

These results prove the effectiveness of the principles that we propose for automatic tuning of the computer architecture.

Frequently generated instruction patterns

Table I is a list of the instructions frequently generated in maximum execution efficiency strategy as well as in maximum economical effect strategy. The detected instruction patterns in the former strategy consist of more machine instructions than those in the latter strategy.

The meaning of most of the long instruction patterns is indicated in the table. For instance, the instruction pattern...
Number of Instruction Patterns Implemented in Firmware

(a) Maximum Execution Efficiency Strategy (Select the instruction patterns so that I becomes maximal.)

(b) Maximum Economical Effect Strategy (Select the instruction patterns so that $\mu/e$ becomes maximal.)

Figure 8—Tuning results (bubble sort problem on the HP-2100 PASCAL machine).

Figure 9—Tuning results (bubble sort problem on the B-1700 PASCAL machine).

Table I — List of the Instruction Frequently Generated
(Bubble Sort Problem on the HP-2100 PASCAL Machine)

<table>
<thead>
<tr>
<th>No.</th>
<th>Pattern No.</th>
<th>Instruction Patterns</th>
<th>Meanings</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Maximum execution efficiency</td>
</tr>
<tr>
<td>1</td>
<td>88</td>
<td>LAO LDO CHK DEC IXA</td>
<td>IND</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>LDO LDO LEQ FJP</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>40</td>
<td>LDO INC SRO UJP</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>70</td>
<td>LDO SRO</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>41</td>
<td>LAO LDO CHK DEC IXA</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>19</td>
<td>LDO STO</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>SRO LDC STR</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No.</th>
<th>Pattern No.</th>
<th>Instruction Patterns</th>
<th>Meanings</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Maximum economical effect</td>
</tr>
<tr>
<td>1</td>
<td>17</td>
<td>DEC IXA</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>LDO CHK</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>LDO LDO</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>31</td>
<td>LDO INC SRO</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>LEG FJP</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>70</td>
<td>LDO SRO</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>73</td>
<td>IND SRO</td>
<td></td>
</tr>
</tbody>
</table>
tages, since the possibility of handling multiple operands is not taken into account. It is desirable to make it possible to expand the operand field at the microprogramming level. The microprogrammable bit addressing function incorporated in the B-1700 and the machine instructions concerning the manipulation of multiple operands provided in the PDP VAX-11/78011 make this problem easier to handle.

2. If the tuning is performed using both instruction patterns and their corresponding address patterns, a better tuning effect is achieved. For instance, the broken line in Figure 11 indicates the tuning results using both the instruction patterns and the address patterns. This result was obtained by a manual procedure. However, it is possible to automate the tuning procedure using address patterns by modifying the proposed algorithm.

**IMLs and host computers**

We arbitrarily selected two IMLs, namely the PASCAL and HP-2100 machine instructions. Nevertheless, significant performance improvement is achieved. Therefore, if we would use an IML instruction set which is more suitable for tuning procedure, an even better performance achievement is to be expected.

The HP-2100 is intentionally designed for the machine instructions. Therefore, it does not have effective features for tuning. Although the B-1700 is an emulation-oriented computer, it seems to us that its microprogram capability is designed to integrate the hardware and software on the Master Control Program. Therefore, some of the well organized mechanisms were not utilized in the automatic tuning procedure.

We think an automatic tuning oriented computer should be developed in the future.

**Tuning curve and learning**

The tuning curve is shown in Figures 12 and 13. The experimental results confirm that the methods to estimate the tuning effect are useful for the proposed algorithm.
As shown in Table I, the frequently generated instruction patterns are significant. These patterns may often be generated in other application programs. This proves that the proposed method is very effective to improve IMLs for a wide variety of applications.

CONCLUSION

Development of a computer architecture that is adaptable to the problems to be solved is an important area of research. In this paper, we consider an automatic tuning of computer architectures at the ISP level using dynamic microprogramming techniques. After a brief explanation of the principles of the automatic tuning mechanisms, a detailed algorithm for the tuning procedure is described.

The basic processes of the proposed automatic tuning algorithm are:

1. Monitoring of the dynamic characteristics of the program to be solved.
2. Analysis of the monitored information to create new instructions.
3. Feedback of corresponding microprogrammed instructions to the WCS in the computer to form a new enriched architecture.
4. A learning process with regard to the tuning behavior to guarantee faster achievement of performance improvement.

The first three processes are automatically carried out until the desired performance improvement is achieved.

In order to prove the effectiveness of the proposal algorithm, we carried out some experiments on a HP-2100 and a Burroughs B-1700 computer. Our experiments show that tuned programs are executed in 30-60 percent less time than the original programs.

It should be noted that dynamic microprogramming techniques are very effective to dynamically modify the computer architectures during the program execution. Our experiments proved that a computer can change dynamically so as to function optimally depending on the status of the problems to be solved, and that the learning effect is significant. We hope that the results of this research will contribute to the future development of adaptive computer systems and learning machines.

ACKNOWLEDGMENT

We wish to thank Mr. Yamaishi, Mr. Kato and our colleagues of Keio University and the Electrotechnical Laboratory for their kind suggestions and assistance. We would like to gratefully acknowledge the support and counsel of many people of Burroughs Co., Ltd. in Japan.

REFERENCES


APPENDIX

An example of the tuning procedure on the B-1700 PASCAL machine

The following steps should be referred to the tuning algorithm shown in Figure 4.
1. Original PASCAL Program

? SORT AN ARRAY OF INTEGER ?
CONST N=25;
VAR I,J,K,M: INTEGER;
A: ARRAY [1..N] OF INTEGER;
BEGIN FOR I:=1 TO N DO A[I]:=I;
FOR I:=1 TO N-1 DO
BEGIN K:=I; M:=A[I];
FOR J:=I+1 TO N DO
IF A[I]>M THEN
BEGIN K:=J;
M:=A[J]
END;
A[K]:=A[I]; A[I]:=M
END.

2. Divide program into blocks at IML instruction level.

MST 0
· CUP SORT
STP

SORT ENT 36
LDCI 1
SRO I
LDCI .WK1

.L01 LDO 1
LOD .WK1
LEQI
FJP .L02

LAO A
LDO I
CHK
DEC I
IXA 1
LDO I
STO
LDO I
INC I
SRO I
UJP .L01

3. Extract possible instruction patterns from each block.

<table>
<thead>
<tr>
<th>Pattern No.</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MST CUP</td>
</tr>
<tr>
<td>2</td>
<td>LDCI SRO</td>
</tr>
<tr>
<td>3</td>
<td>SRO LDC</td>
</tr>
<tr>
<td>4</td>
<td>LDC STR</td>
</tr>
<tr>
<td>5</td>
<td>LDC SRO LDC</td>
</tr>
<tr>
<td>6</td>
<td>SRO LDC STR</td>
</tr>
<tr>
<td>7</td>
<td>ENT LDC SRO LDC STR</td>
</tr>
<tr>
<td>8</td>
<td>LDO LOD</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block No.</th>
<th>Pattern No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2 3 4 5 6 7</td>
</tr>
<tr>
<td>3</td>
<td>8 9 10 11 12 13</td>
</tr>
<tr>
<td>4</td>
<td>14 15 16 17 18 19 20 21 22 23</td>
</tr>
<tr>
<td></td>
<td>24 25 26 27 28 29 30 31 32 33</td>
</tr>
<tr>
<td></td>
<td>34 35 36 37 38 39 40 41 42 43</td>
</tr>
<tr>
<td></td>
<td>44 45 46 47 48 49 50 51 52 53</td>
</tr>
<tr>
<td></td>
<td>54 55 56 57 58 59 60 61 62 63</td>
</tr>
<tr>
<td></td>
<td>64 65 66 67 68</td>
</tr>
<tr>
<td>5</td>
<td>2 3 5 69</td>
</tr>
<tr>
<td>6</td>
<td>8 9 10 11 12 13</td>
</tr>
</tbody>
</table>

4. Calculate the length and the execution time of each pattern.

<table>
<thead>
<tr>
<th>Pattern No.</th>
<th>Pattern</th>
<th>Length</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MST CUP</td>
<td>2</td>
<td>25.1</td>
</tr>
<tr>
<td>2</td>
<td>LDCI SRO</td>
<td>2</td>
<td>21.5</td>
</tr>
<tr>
<td>3</td>
<td>SRO LDC</td>
<td>2</td>
<td>21.5</td>
</tr>
<tr>
<td>4</td>
<td>LDC STR</td>
<td>2</td>
<td>22.7</td>
</tr>
<tr>
<td>5</td>
<td>LDC SRO LDC</td>
<td>2</td>
<td>30.2</td>
</tr>
<tr>
<td>6</td>
<td>SRO LDC STR</td>
<td>2</td>
<td>35.6</td>
</tr>
<tr>
<td>7</td>
<td>ENT LDC SRO LDC STR</td>
<td>2</td>
<td>50.0</td>
</tr>
<tr>
<td>8</td>
<td>LDO LOD</td>
<td>2</td>
<td>22.5</td>
</tr>
</tbody>
</table>

5. Measure execution frequencies of each block.

<table>
<thead>
<tr>
<th>Block No.</th>
<th>Frequency [f]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

From the collection of the Computer History Museum (www.computerhistory.org)
6. For each pattern, calculate execution frequencies then calculate weight.

<table>
<thead>
<tr>
<th>Pattern No.</th>
<th>Frequency [f]</th>
<th>Weight [g]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1.02 x 10^{-4}</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1.75 x 10^{-4}</td>
</tr>
<tr>
<td>3</td>
<td>26</td>
<td>2.28 x 10^{-3}</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>2.31 x 10^{-2}</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1.64 x 10^{-4}</td>
</tr>
<tr>
<td>6</td>
<td>25</td>
<td>2.42 x 10^{-3}</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>8.14 x 10^{-5}</td>
</tr>
<tr>
<td>8</td>
<td>375</td>
<td>3.93 x 10^{-2}</td>
</tr>
<tr>
<td>9</td>
<td>375</td>
<td>4.21 x 10^{-2}</td>
</tr>
<tr>
<td>10</td>
<td>375</td>
<td>3.44 x 10^{-2}</td>
</tr>
</tbody>
</table>

Note: \( \xi(p) = \frac{T(p)}{T_{o} \times \gamma(p)} \)
\( T(p) = r(p) \times T(p) \)
\( T_{o} = T(1) + T(13) + T(40) + T(68) + T(69) + T(95) + T(96) + T(97) + T(98) \)
\( = 1.23 \times 10^{4} \mu s \)

7. Calculate \( \dot{\theta}, \mu \) and \( \dot{\mu}/\dot{\theta} \) from Equations 4 and 5. In the case that none of the candidates is in the tuning data base, but the coefficients for Equations 4 and 5 are registered.

\[ A = 0.69 \quad b = 4.6 \quad \dot{\theta}(p) = b \times \gamma(p) + c \]
\[ m = 1.14 \quad c = 14.7 \quad \mu(p) = A \times \dot{\theta}(p)^{m} \times \xi(p)^{n} \]
\[ n = 0.61 \]

<table>
<thead>
<tr>
<th>Pattern No.</th>
<th>[\dot{\theta}]</th>
<th>[\mu]</th>
<th>[\dot{\mu}/\dot{\theta}]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24.1</td>
<td>0.1</td>
<td>0.004</td>
</tr>
<tr>
<td>2</td>
<td>24.1</td>
<td>0.1</td>
<td>0.006</td>
</tr>
<tr>
<td>3</td>
<td>24.1</td>
<td>0.6</td>
<td>0.03</td>
</tr>
<tr>
<td>4</td>
<td>24.1</td>
<td>0.6</td>
<td>0.03</td>
</tr>
<tr>
<td>5</td>
<td>28.7</td>
<td>0.2</td>
<td>0.005</td>
</tr>
<tr>
<td>6</td>
<td>28.7</td>
<td>0.8</td>
<td>0.03</td>
</tr>
<tr>
<td>7</td>
<td>37.9</td>
<td>0.1</td>
<td>0.004</td>
</tr>
<tr>
<td>8</td>
<td>24.1</td>
<td>3.6</td>
<td>0.15</td>
</tr>
<tr>
<td>9</td>
<td>24.1</td>
<td>3.8</td>
<td>0.16</td>
</tr>
<tr>
<td>10</td>
<td>24.1</td>
<td>3.3</td>
<td>0.14</td>
</tr>
</tbody>
</table>

8a. Strategy 1—Maximum efficiency improvement

| Sorting result of patterns in descending order of \( \dot{\mu} \). |
|------------------|------------------|
| No. | Pattern No. | [\dot{\theta}] | [\mu] |
| 1   | 88           | 10.6            | 42.5  |
| 2   | 41           | 9.9             | 37.9  |
| 3   | 84           | 9.7             | 37.9  |
| 4   | 97           | 9.6             | 56.3  |
| 5   | 34           | 9.3             | 33.3  |

The rest is omitted but same as in Strategy 2.

8b. Strategy 2—Maximum economical effect

| Sorting result of patterns in descending order of \( \dot{\mu}/\dot{\theta} \). |
|------------------|------------------|
| No. | Pattern No. | [\dot{\mu}/\dot{\theta}] | [\dot{\mu}] | [\dot{\theta}] |
| 1   | 17          | 0.33             | 7.9          | 24.1          |
| 2   | 72          | 0.31             | 7.5          | 24.1          |
| 3   | 26          | 0.30             | 8.5          | 28.7          |
| 4   | 76          | 0.28             | 8.1          | 28.7          |
| 5   | 34          | 0.28             | 9.3          | 33.3          |

...Pattern with the maximum value of \( \dot{\mu}/\dot{\theta} \).

17 DEC IXA

9. Once more perform static analysis of the program to check the change in execution frequencies of the pattern, provided that new instructions are synthesized from the selected patterns.

No. | Pattern | [\dot{\mu}/\dot{\theta}] | [\dot{\mu}] | [\dot{\theta}] |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>DEC</td>
<td>36</td>
<td>STP</td>
<td>Block 1</td>
</tr>
</tbody>
</table>

10. If the total sum of \( \dot{\mu} \) for the candidates of patterns exceeds the value of \( \dot{\theta} \) or if the total sum of \( \dot{\theta} \) for the candidates of patterns exceeds the value of \( \dot{\mu} \) then go to 11, else go to 6.

11. Original PASCAL IML interpreter written in Burroughs B-1700 MIL.
% IML FETCH AND DECODE
CLEAR BASE.REG
MOVE 24 TO CP
MOVE A TO FETCH.ADDRESS
%A=ACTUAL ADDRESS OF FETCH
FETCH
MOVE FETCH.ADDRESS TO TAS
MOVE PC TO T
SHIFT T LEFT BY 5 BITS TO X
SHIFT T LEFT BY 4 BITS TO Y
MOVE SUM TO FA
ADD BASE.REG TO FA
FA.POINTS TO IML
READ DL(OP)+DL(P) BITS TO T
PT.FA Q
READ DL(Q) BITS TO X
MOVE X TO Q.FIELD
EXTRACT DL(OP) BITS FROM T(10)
TO Y
MOVE PC TO L
COUNT L UP BY 1
MOVE L TO PC
MOVE Y TO M
JUMP FORWARD
GO TO LOD
GO TO LDO
DEC
CALL POP.UP.BREG.1
MOVE Q.FIELD TO Y
MOVE DIFF TO BREG.1
GO TO PUSH.DOWN.BREG.1
IXA
CALL POP.UP.BREG.1
MOVE Q.FIELD TO AREG.1
IXA.1
CALL SET.SIGN.AND.ABS
MOVE AREG.1 TO Y
MOVE BREG.1 TO FA
CALL I.MULTIPLY
IF SIGNS.ARE.DIFFERENT THEN BEGIN
CLEAR X
MOVE DIFF TO Y
END
MOVE Y TO TAS
CALL POP.UP.BREG.1
MOVE TAS TO Y
MOVE SUM TO BREG.1
GO TO PUSH.DOWN.BREG.1

12. Implement the selected pattern in firmware.

SYN01 % DEC Q1-IXA Q2
% " + - - - - - - + - - - + "
% I SYN01 I I
% " + - - - + - - - - + "
% I I Q 1 I
% " + - - - - - - - - + "
% I I Q 2 I
% " + - - - - - - - - + "
% I DUMMY I
% " + - - - - - - - - +"
% BUMP(PC,1)
READ 24 BITS TO X
MOVE X TO AREG.1
CALL POP.UP.BREG.1
MOVE Q.FIELD TO Y
MOVE DIFF TO BREG.1
% BREG.1=TOP OF STACK
% AREG.1=Q2
GO TO IXA.1

13. Measure the efficiency improvement by means of the new IML instructions then register the patterns corresponding to the new IML instructions in the tuning data base, and update the tuning curve.

<table>
<thead>
<tr>
<th>Tuning result for Pattern 17</th>
<th>Routine Name</th>
<th>Cycle</th>
<th>Routine Name</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>FETCH</td>
<td>25</td>
<td>FETCH</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>48</td>
<td>SYN01</td>
<td>276</td>
<td></td>
</tr>
<tr>
<td>FETCH</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IXA</td>
<td>265</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>363</strong></td>
<td></td>
<td></td>
<td><strong>Total</strong></td>
</tr>
</tbody>
</table>

Execution Time \([\tau] 60.4 \mu s\) Execution Time \([\tau'] 50.3 \mu s\)
Required WCS \([\theta] 9\) words

\[
\mu(17) = \frac{(\tau(17)-\tau'(17)) \times f(17)}{T_0} \times 100\% \\
= 4.8\% 
\]