COMPUTATIONAL COMPLEXITY OF RANDOM ACCESS STORED PROGRAM MACHINES

J. Hartmanis

Technical Report

No. 70-70

August 1970

Department of Computer Science
Cornell University
Ithaca, New York 14850

COMPUTATIONAL COMPLEXITY OF RANDOM ACCESS STORED PROGRAM MACHINES

J. Hartmanis

ABSTRACT.

In this paper we explore the computational complexity measure defined by running times of programs on random access stored program machines, RASP's. The purpose of this work is to study more realistic complexity measures and to provide a setting and some techniques to explore different computer organizations. The more interesting results of this paper are obtained by an argument about the size of the computed functions. For example, we show (without using diagonalization) that there exist arbitrarily complex functions with optimal RASP programs whose running time cannot be improved by any multiplicative constant. We show, furthermore, that these optimal programs cannot be fixed procedures and determine the difference in computation speed between fixed procedures and self-modifying programs. The same technique is used to compare computation speed of machines with and without built in multiplication. We conclude the paper with a look at machines with associative memory and distributed logic machines.

INTRODUCTION.

The purpose of this paper is to study the quantitative problems about computation speed of random access stored program machines, called RASP's. These abstract machines were defined [1,2] to provide theoretical models which look more like real computers and which reflect some aspects of real computing more directly than previously studied Turing machines.

In the following section we give an informal definition of random access stored program machines and derive some elementary results about the computational complexity measure based on the number of operations used by RASP's. Next, by a new technique involving the size of the computed functions, we study the problem of how well we can define and approach the best possible computation time with RASP programs. It is shown that there exist arbitrarily complex computations for which we can write programs whose computation time cannot be improved by any factor. That is, there exist functions $f_i(n)$ which can be computed in time $T_i(n)$ but cannot be computed in time $(1 - \varepsilon)T_i(n)$ for any $\varepsilon > 0$. This result contrasts sharply the corresponding results for time-limited, tape-limited or reversal-limited Turing machine computations [3,4, 5,6] all of which can be improved by a constant factor. It is also interesting to observe that this result is obtained without using a diagonal argument which up until now was the only technique to prove results of this type. Clearly, some results about computation speeds have been obtained previously by counting arguments but they did not extend to arbitrarily complex computations [7,8].

Further analysis of the above result reveals that these optimal programs cannot be fixed procedures in that they must modify themselves and grow in length. This proves that for infinitely many computations fixed procedures are slower than self-modifying programs. Furthermore, it is shown that self-modification can at best speed-up the computation time by a constant factor and that for infinitely many, arbitrarily complex computations they are faster by a factor which depends on the size of the fixed program. Next, we compare RASP's with and without built in multiplication and prove that the RASP's with multiplication are faster and give an upper bound for the maximal speed difference. It is interesting to note that the above mentioned results are much sharper than any results we have been able to obtain for Turing machine computations. For example, it is still not known whether million-tape Turing machines are faster than one-tape Turing machines for complicated computations [9]. We only know that for real-time computations two tapes are better than one tape [7,8]. Similarly we do not know whether non-deterministic Turing machines are faster than deterministic machines nor do we know that Turing machines with jump operations or multidimensional tapes have any speed advantage over regular, one-tape machines for arbitrarily complex computations.

We conclude the paper by showing that RASP's with associative memories can at best improve the computation speed by the square root and raise some questions about distributed logic machines.

RANDOM ACCESS STORED PROGRAM MACHINES

In this section we describe a random access stored program machine model, RASP, and derive some results about computation times on RASP's. We are giving only an informal description of our model to avoid making a rather simple topic look more complicated.

A random access stored program machine, abbreviated RASP, consists of a memory M and a finite set of instructions I ,

$$\mathrm{RASP} = \langle M, I \rangle .$$

The memory M of the RASP consists of two special registers, called Accumulator, AC , and Instruction Counter, IC , and an infinite sequence of memory registers:

$$R1 , R2 , R3 , \ldots$$

each register is capable of holding an arbitrary, finite length binary string. The content of the registers will be denoted by <AC> , <IC> , and <Rn> or <n> , respectively.

The non-empty, finite set of instructions can contain the three types of instructions:

operations,

conditional transfers,

store instructions,

and must contain a HALT instruction.

A. The operations are denoted by

$$F_i \ , \ F_i,n \ , \ F_i,\langle n\rangle \quad \text{and} \quad F_i,\langle\langle n\rangle\rangle$$

where F_i in the first case is a (name of a) one-variable recursive function and the effect of executing F_i is to replace the content of AC by $F_i[\langle AC\rangle]$. In the three remaining cases F_i denotes a two-variable recursive function and the effect of executing F_i,n , $F_i,\langle n\rangle$ and $F_i,\langle\langle n\rangle\rangle$ is to replace the content of AC by

$$F_i[\langle AC\rangle,n] \ , \ F_i[\langle AC\rangle,\langle n\rangle]$$

and

$$F_i[\langle AC\rangle,\langle\langle n\rangle\rangle] = F_i[\langle AC\rangle,\langle R\langle n\rangle\rangle] \ ,$$

respectively. The execution of an operation also replaces $\langle IC\rangle$ by $\langle IC\rangle + 1$.

B. The conditional transfers are denoted by

$$P_i,n \quad \text{and} \quad P_i,\langle n\rangle$$

where P_i is (the name of) a recursive predicate. The execution of P_i,n and $P_i,\langle n\rangle$ transfers the control to Rn and $R\langle n\rangle$, respectively, if

$$P_i[\langle AC\rangle] = 1 \ ,$$

otherwise it transfers control to the next instruction. For the sake of convenience, we will later also use two variable recursive predicates

$$P_i,m,n \quad \text{or} \quad P_i,\langle m\rangle,\langle n\rangle$$

which will transfer control to Rn and R<n> , respectively, provided

$$P_i[\langle AC\rangle,m] = 1 \quad \text{or} \quad P_i[\langle AC\rangle,\langle m\rangle] = 1 \ ,$$

otherwise the control is transfered to the next instruction.

C. The store instructions are denoted by

STO,n or STO,<n>

The execution of STO,n copies the contents of AC into the register Rn and the execution of STO,<n> copies the contents of AC into the register R<n> (where the binary string stored in Rn is interpreted as an integer). The execution of the store instruction also replaced <IC> by <IC> + 1 .

There is also a special instruction denoted by HALT . When the control is transfered to this instruction no further instructions are executed (since <IC> is not changed).

Without loss of generality we assume that the integers have the standard binary encoding: 0 , 1 , 10 , 11 , 100 , ... The instructions are represented by sequences starting with say, a double zero and with indications whether they operate on addresses or indirect addresses and proper delineation of where the data part starts (i.e. n or <n>).

A program consists of a finite sequence of instructions and data (binary strings not representing instructions) stored in the memory. We also assume that at the start of the computation <IC> = 1 .

The input to the machine is the integer placed in AC at the start of the computation and the result is the integer in AC after the computation reaches a HALT instruction.

The program is executed by executing the instructions in the order indicated by the instruction counter starting with <IC> = 1 until the machine reaches a HALT instruction, that is <<IC>> = HALT .

The machine jams if some undefined situation is entered and we do not consider this as yielding a result.

From the description of RASP's we can easily describe their computational power.

Lemma: Any function computed by a RASP program is a partial recursive function and there exists a RASP on which we can compute all partial recursive functions.

Proof: Any RASP computation can be simulated on a Turing machine and therefore they compute only partial recursive functions. Since there exist RASP's which can be programmed to simulate any Turing machine we see that there exist universal RASP's.

We say that a RASP program is a fixed program if the instructions of the program are not changed during the execution for any input and if only the instructions of the original program are executed.

Thus the program may write another program during the computation but it is a fixed program as long as it did not alter its original instructions or execute any of the instructions in the program it wrote.

It should be noted that it is recursively undecidable whether a given program for a universal RASP is fixed. At the same time one can easily prove the next result.

Lemma: There exist universal RASP's which have recursive sets of fixed programs to compute all partial recursive functions.

In this paper we are only interested in universal RASP's and we can see that it is easy to specify small sets of instructions which are sufficient to obtain universality in RASP's. At the same time we are not primarily interested in the "small universal" RASP's and we will not dwell on the minimal number of instructions needed for universality.

We observe though that any universal RASP can be used to define a computational complexity measure [10] if the step-counting function $T_i(n)$ associated with the i-th program P_i is given by the number of operations performed by the RASP on program P_i before halting on input n. If the machine does not halt or jams then $T_i(n)$ is undefined.

<u>Lemma</u>: The number of instructions executed by a universal RASP before halting defines a computational complexity measure.

<u>Proof</u>: We observe that since our RASP is universal we have a list of algorithms (programs) P_1, P_2, P_3, ... to compute all partial recursive functions. Furthermore, the step-counting function $T_i(n)$ associated with program P_i is defined if and only if the program P_i is defined (halts) on input n and we can test recursively whether $T_i(n) \leq k$ for all k. Thus we have a computational complexity measure as defined in [10].

It should be observed that there exist universal RASP's which use only a finite number of registers and therefore the number of registers used by a universal RASP does not always define a computational complexity measure.

On the other hand, the number of <u>conditional statements</u> executed by any universal RASP before halting does define a computational complexity measure which deserves further investigation and is not studied in this paper.

The above lemma immediately implies that the computational complexity measures defined by the running times of universal RASP's satisfy the many results proven for abstract complexity measures [10,11,12]. On the other hand, these general results give no specific information about RASP computations nor do they give any relations between the computation speeds of different RASP's, nor their relation to running speeds on other types of machines. Thus our next task will be to study the computation times of specific RASP's and the relations between computation times of different RASP's.

To give more concreteness to our proofs, which will be generalized later, we now define a specific RASP, denoted by RASP1. The machine has the following instructions:

NAME	MEANING
TRA,n TRA,<n>	transfer control to register Rn and R<n> , respectively, i.e. <IC> = n and <IC> = <n> , respectively.
TRZ,n TRZ,<n>	if <AC> = 0 , transfer control to register Rn and R<n>, respectively, otherwise continue to next instruction.
STO,n STO,<n>	Store <AC> in Rn and R<n> , respectively (the content of AC is not altered).
CLA,n CLA,<n> CLA,<<n>>	n , <n> , and <<n>> , respectively, is stored in AC (the content Rn and R<n> is not altered).
ADD,n ADD,<n> ADD,<<n>>	<AC> is replaced by <AC> + n , <AC> + <Rn> and <AC> + <R<n>> , respectively.
SUB,n SUB,<n> SUB,<<n>>	<AC> is replaced by <AC> ∸ n , <AC> ∸ <Rn> and <AC> ∸ <R<n>> , respectively.
HALT	No further instructions are executed.

SIZE ARGUMENTS IN RASP COMPUTATIONS.

A general problem for specific complexity measures it to determine how sharply we can bound the computation time of functions in this measure. In this section we use an argument about the size of the computed functions to show that for RASP1 and many other RASP's we can bound the computation times very sharply indeed. As a matter of fact, we show that for any $\varepsilon > 0$ there exist arbitrarily complex functions f_i which can be computed by fixed RASP1 programs in time $T_i(n)$ but cannot be computed by any RASP1 program in time $(1 - \varepsilon)T_i(n)$.

For programs which can modify themselves (and grow) during the computation, we obtain even a sharper result. We show that there exist arbitrarily difficult functions which can be computed in time $T(n)$ by a self-modifying RASP1 program but cannot be computed by a RASP1 program in time $(1 - \varepsilon)T(n)$ for any $\varepsilon > 0$.

This result is then furthermore used to show that self-modifying programs are inherently faster than fixed programs. It is interesting to note that these results show that RASP computation times behave quite differently from Turing machine computation times which always have a constant speed-up for complex computations [3].

The basic idea in the proof that RASP1 programs cannot be speed-up by any constant factor is that we have operations which increase the size of the contents of AC by more than any other operations. Thus we just observe that repeating these operations increases the value of the computed function faster than any other sequence of operations and that this sequence of operations cannot be substantially shortened.

We first observe that for any k in $2k$ operations on RASP1 we can compute the function

$$F(n) = n2^{k}$$

by the following program with $\ell \geq 2k + 1$:

```
STO,ℓ
ADD,<ℓ>
STO,ℓ
ADD,<ℓ>
STO,ℓ
ADD,<ℓ>
 .
 .
 .
STO,ℓ
ADD,<ℓ>
HALT  ,
```

where the first pair of operations is repeated k times.

This program can be slightly modified but a simple proof by induction shows that the number of operations cannot be decreased, for sufficiently large values of n . Thus we obtain the following result.

Lemma: The computation of the function

$$F(n) = n2^k$$

on RASP1 requires at least $2k$ operations for large values of n.

We now use this result to show that RASP1 programs cannot be sped-up.

Lemma: For any $\varepsilon > 0$ there exists a fixed RASP1 program with running time $T(n)$ such that no other program can compute the same function in time $(1 - \varepsilon)T(n)$.

Proof: Consider the following program which computes

$$F(n) = n2^{kn+1}$$

```
R1        STO,2k + 11
R2        STO,2k + 10
R3        ADD,<2k + 10>
          STO,2k + 10
          ADD,<2k + 10>
          .
          .
          .
          STO,2k + 10
R(2k+1)   ADD,<2k + 10>
          STO,<2k + 10>
          CLA,<2k + 11>
          TRZ,<2k + 9>
          SUB,1
          STO,<2k + 11>
          CLA,<2k + 10>
          TRA,3
R(2k+9)   HALT
```

For $n > 1$ the running time $T(n)$ of this program satisfies the inequality,

$$T(n) < (2k + 9)(n + 1) .$$

On the other hand, be a previous lemma, we know that this function cannot be computed by any program with running time $t(n)$ such that

$$t(n) < 2kn .$$

If we choose the integer k such that $k > \frac{1}{\varepsilon}$. Then obviously for large values of n

$$(1 - \varepsilon)T(n) < (1 - \varepsilon)(2k + 9)(n + 1) < 2kn .$$

Thus

$$(1 - \varepsilon)T(n) < 2kn ,$$

which shows that the computation time $(1 - \varepsilon)T(n)$ is not sufficient to compute the desired function by any RASP1 program.

Next we show how our result can be generalized to arbitrarily complex computations.

Theorem: For any $\varepsilon > 0$ there exist arbitrarily complex functions f_i which can be computed in time $T_i(n)$ by fixed RASP1 programs but cannot be computed by any RASP1 program in time $(1 - \varepsilon)T_i(n)$.

Proof: We know that there exist arbitrarily complex functions $g_i(n)$ which can be computed in $g_i(n)$ operations. Our desired program will compute

$$f_i(n) = n2^{kg_i(n)} ,$$

for a properly choosen k which depends on the given ε. The program will first compute $g_i(n)$ and then cycle $g_i(n)$ times, through $2k$ instructions of doubling k times the content of the accumulator. Since $g_i(n)$ can be computed in $g_i(n)$ operations we see that the computation time $T_i(n)$ of

$$f_i(n) = n2^{kg_i(n)}$$

satisfies

$$T_i(n) \leq (2k + 10)g_i(n) .$$

Since $g_i(n)$ operations suffice to compute $g_i(n)$ and nine operations on each cycle suffice to decrement $g_i(n)$ by one each time. Recalling that the computation of $f_i(n)$ requires at least $2kg_i(n)$ operations and choosing $k > \frac{1}{\varepsilon}$, we see that for large n

$$(1 - \varepsilon)T_i(n) \leq (1 - \varepsilon)(2k + 10)g_i(n) < 2kg_i(n) .$$

Thus $f_i(n)$ cannot be computed in $(1 - \varepsilon)T_i(n)$ operations, as was to be shown.

It is interesting to observe that the previously used size arguments can also be applied to computational complexity measures based on RASP (or RAM) operations which are weighted by the size of the arguments or the size of the result. Such measures have been defined and studied by S.A. Cook and we can show that for many weighting functions these measures also have no speed-up [2].

SELF-MODIFYING RASP PROGRAMS

Next we show that for self-modifying programs we can get even sharper time bounds. Note that for any $\varepsilon > 0$ we exhibited infinitely many fixed RASP programs whose computation time cannot be speeded-up by the factor $(1 - \varepsilon)$. For self-modifying programs we will show that the ε does not have to be given, in that there exist infinitely many programs whose running time cannot be improved by the factor $(1 - \varepsilon)$ for any $\varepsilon > 0$.

The basic idea of the proof is very simple. With increasing n our program will write new programs with longer and longer runs of

```
STO,ℓ
ADD,<ℓ>
```

instructions and thus have relatively fewer conditional statements. From this we will be able to show that the running-times of these programs approach with increasing n the order of the optimal time and therefore they cannot be speeded-up by any constant factor. After that we show that self-modification is essential for this result.

We start with an illustrative example. Consider the function

$$f(n) = n \cdot 2^{n^2} .$$

We know that any program computing $f(n)$ cannot use less than $2n^2$ operations for large n. On the other hand, consider the following RASP1 program P for f:

a) for input n P writes in successive memory registers n times the pair of instructions

$$\begin{array}{l} \mathrm{STO}, \ell \\ \mathrm{ADD}, \langle \ell \rangle \end{array}$$

and the necessary control statements to store n and decrement that register by 1 each time it cycles through the STO-ADD sequence.

b) After that P initiates the execution of this new program.

We see that the writing of the new program does not have to take more than cn operations for $n \geq 1$. The execution of the new program requires no more than

$$(2n + 20)n \text{ - operations.}$$

Thus

$$T(n) \leq 2n^2 + 20n + cn$$

and for any $\varepsilon > 0$ and sufficiently large n

$$(1 - \varepsilon)T(n) < 2n^2 ,$$

showing that this computation cannot be speeded-up by any constant factor. We summarize our conclusion.

Lemma: There exists a RASP1 program for the computation of

$$f(n) = n2^{n^2}$$

with running time $T(n)$ and no RASP1 program can compute f in time $(1 - \varepsilon)T(n)$ for any $\varepsilon > 0$.

Just as in the previous section we can extend this result to arbitrarily complex functions.

Theorem: There exist arbitrarily complex functions $f_i(n)$ which can be computed in time $T_i(n)$ on RASP1 but cannot be computed in time $(1 - \varepsilon)T_i(n)$ for any $\varepsilon > 0$.

Proof: The previous proof goes through if we replace in the new program the n STO-ADD instructions by $g_i(n)$ STO-ADD instrictions, where $g_i(n)$ can be computed in $g_i(n)$ operations. Then the program computes

$$f_i(n) = n \cdot 2^{g_i^2(n)}$$

in time

$$T_i(n) \leq (2g_i(n) + c)g_i(n)$$

and we see that this time cannot be speeded-up by a factor $(1 - \varepsilon)$ for $\varepsilon > 0$.

The next result shows that there exist computations for which self-modifying programs are faster than fixed programs.

<u>Theorem</u>: There exist arbitrarily complex functions $f_i(n)$ with self-modifying programs running in time $T_i(n)$ such that any fixed program for f_i of length ℓ cannot run faster than $(1 + \frac{1}{2\ell})T_i(n)$ for large n.

<u>Proof</u>: Consider the functions

$$f_i(n) = n \cdot 2^{g_i^2(n)}$$

of the previous proof, which we know to have running times

$$T_i(n) \leq (2g_i(n) + c)g_i(n) .$$

Let P be any fixed program of length ℓ for $f_i(n)$. Then this program must execute at least one conditional statement every ℓ operations. Furthermore, to compute $f_i(n)$ the program must execute at least

$$2g_i^2(n)$$

operations which are not conditional statements. Thus the running time of P , T(n) , must be such that

$$T(n) > 2g_i^2(n) + \frac{2g_i^2(n)}{\ell} = 2g_i^2(n)(1 + \frac{1}{\ell})$$

for large n . Thus we conclude that

$$(1 + \frac{1}{2\ell})T_i(n) \leq (1 + \frac{1}{2\ell})(2g_i^2(n) + cg_i(n)) < (1 + \frac{1}{\ell})2g_i^2(n)$$

and therefore

$$(1 + \frac{1}{2\ell})T_i(n) \leq T(n) ,$$

for large n . This concludes the proof by showing that the fixed program is slower by the factor $(1 + \frac{1}{2\ell})$.

A somewhat more detailed argument allows us to replace the factor $(1 + \frac{1}{2\ell})$ by $(1 + \frac{1}{\ell})$.

The previous results shows that there exist computations for which any fixed program can be replaced by another fixed program which runs faster by a positive factor. Furthermore, there is a self-modifying program which runs faster than any fixed program and whose running-time cannot be improved by a multiplicative factor. Next we look at the problem of determining how much computation speed can be gained by going from fixed to self-modifying programs.

We will show that for RASP's with sufficiently rich instruction sets we can easily replace self-modifying programs by fixed programs and loose no more than a constant factor in computing time, where the factor depends only on the instruction set of the machine. On the other hand, as it will be seen from a later discussion, the situation for RASPs with very simple instruction sets is not well undestood and there are several open problems. Similarly, we will show that by permitting operations defined on constants (not on contents of registers) we can define RASP's for which fixed programs are arbitrarily slower than self-modifying ones.

The main problem in replacing arbitrary programs by fixed programs is to determine quickly what a self-modifying program does when it executes an operation stored in Rn . If we can do this then we can load the data in an appropriate register and execute the corresponding instruction on this data in our fixed program. Thus simulating the self-modifying program by the operations of the fixed program and storing the results in the appropriate places of the simulated self-modifying program.

To be able to do this simulation fast we add three new types of instructions to RASP1:

TFX,n,m TFX,<n>,<m>	if <AC> has prefix n or <Rn> , transfer control to register Rn or <Rm> , respectively. Otherwise go to next instruction.

PFX,n PFX,<n>	the string n or contents of Rn , respectively, are prefixed to <AC>
DFX,n DFX,<n>	if <AC> has n or <Rn> as prefix, respectively, then this prefix is removed from <AC> ,

and we denote a RASP1 with these additional instructions by RASP2.

Theorem: There exists a constant c such that any RASP2 program P with running time $T(n)$ can be replaced by an equivalent fixed program P' with running time $T'(n)$ satisfying the inequality

$$cT(n) \geq T'(n)$$

Proof: We give only an outline of the proof. The basic idea is that we will have a fixed program P' which simulates the program P by using only the fixed operations contained in P at the start of the computation. The program P' keeps a copy of the current status of program P and keeps track of the instruction counter content. To simulate one operation of P the program P' transfers the contents of the register containing the next operation of P to the AC . Then, by using the "prefix" instructions it determines what instruction it is and separates the data. After that a corresponding operation in the fixed program P' is used to execute the desired operation on this data and the result is stored in the copy of P which P' manages. In this manner the fixed program P' is "driving" a copy of the program P and computing the same function as P . Since we have the "prefix" operations, we can simulate one operation of P in a fixed number of P' operations. This shows that the program P' requires only constant times more operations than P . Thus for a fixed $c > 0$

$$c \cdot T(n) \geq T'(n)$$

Combining our previous results we get the following.

Corollary: For RASP2 there exist infinitely many arbitrarily complex functions for which the computation time of any fixed program can be improved by some positive factor and for which we have self-modifying programs whose run-times cannot be improved by any factor. Furthermore, there exists a constant factor which limits how much faster self-modifying programs can be than fixed ones.

Another generalization is also immediate.

Theorem: For any RASP which contains all the instructions of RASP2 and contains no operations defined only on constants, there exists a constant factor which limits how much faster self-modifying programs can run than fixed ones.

Next we observe that if a RASP has some operations which are defined only on constants, then the difference in computation time between fixed programs and self-modifying programs can be arbitrarily large. We prove a special case which can easily be generalized.

Let RASP3 be the RASP obtained from RASP1 by deleting the instructions

ADD,<n> and ADD,<<n>>

and adjoining the instruction

$$\mathrm{PFX},x_o$$

where x_o is the part of the code for the ADD,c instruction which comes before c . Thus the execution of

$$\mathrm{PFX},x_o$$

replaces

<AC> = m by ADD,m .

Observe that RASP3 is a universal computer. At the same time any fixed RASP3 program has a maximal constant which it can add to the accumulator. Thus any fixed RASP3 program can at best grow linearly with the number of operations.

On the other hand self-modifying programs can grow faster. Consider the program

R1	STO,7
R2	PFX,x_o
R3	STO,5
R4	CLA,7
R5	

which copies input in R7 , then changes <AC> = n to ADD,n ; stores it in R5 ; places n in AC and executes ADD,n ; thus doubling the input. By building this program in to a larger program we can realize a function which grows exponentially with the number of operations. Thus we have shown the following result.

Lemma: There exist a function whose computation by any fixed RASP3 program requires exponentially more operations than a self-modifying RASP3 program.

Clearly we can generalize this result to obtain arbitrarily large gaps between the running times of fixed programs and self-modifying programs as long as we are permitted to use RASP instructions defined only for constants. It should be realized that this in essence permits the self-modifying program to generate new and more powerful instructions and thus can be considered as a pathology.

Theorem: Let g be an increasing, unbounded, recursive function. Then there exists a RASP and a set of arbitrarily complex functions f_i with self-modifying programs P_i running in time T_i such that any fixed program P_i' for f_i must run in time T_i' satisfying the condition

$$g(T_i(n)) \leq T_i'(n)$$

for sufficiently large n .

We conclude this section by stating several open problems:

1. Can we show that for RASP1 there exist arbitrarily complex zero-one functions whose computation time cannot be speeded-up by any factor?
2. Can we show that for any universal RASP there exist arbitrarily complex functions whose computation time cannot be speeded-up by any factor? Can we prove this for zero-one functions?
3. Can we show that for any universal RASP there exists a recursive set of fixed programs which computes all partial recursive functions?

COMPARISON OF RASP'S AND OTHER MODELS.

In the study of Turing machine running times we have been rather unsuccessful in proving the advantages of additional tapes, multidimensional tapes, jump instructions, etc.

The situation is quite different for RASP's and we illustrate it by showing that RASP1 computations are slower than those performed on RASP1 with the additional instructions

MLT,n
MLT,<n>

whose execution replaces

$$\langle AC\rangle \text{ by } \langle AC\rangle \cdot n \text{ and } \langle AC\rangle \cdot \langle n\rangle ,$$

respectively. We designate these machines as RASP1$^{\times}$

Theorem: There exist arbitrarily complex functions $f_i(n)$ with RASP1$^{\times}$ running times $T_i(n)$ such that any RASP1 program for $f_i(n)$ with running time $T_i'(n)$ must satisfy for large n the condition

$$2^{T_i(n)} \log_2 n < 2T_i'(n) .$$

Proof: We observe that with MLT< > operation we can write programs which in 2k operations reach size

$$f(n) = n^{2^k} .$$

Since a RASP1 can at best compute in $2r$ operations a function of size

$$g(n) = n2^r ,$$

we see that to compute the same function on a RASP1 r must satisfy

$$n2^r \geq n^{2^k} .$$

Thus

$$r \geq (2^k - 1)\log_2 n .$$

This can be generalized to arbitrarily complex functions for which we easily obtain that

$$T_i'(n) > 2^{T_i(n)} \log_2 n ,$$

as was to be shown.

Similarly we can show the differences in computations between many other RASP models, where we utilize the different abilities to build large functions. It turns out that the differences in ability to decrease a register can also be utilized to show differences in computation speeds. Unfortunately we have not been able to extend the last technique to arbitrarily large functions. Thus we have not shown that a machine, with the only subtraction operation SUB,1 , which replaces <AC> by <AC> ∸ 1 , is slower for arbitrarily complex functions than the same machine with the instruction

SUB,<n>

added. Clearly, we can find simple computations for which the second machine is faster. The question remaining open is what happens for arbitrarily complex computations.

Another interesting open problem is to determine whether there exist arbitrarily complex functions in whose computations we can gain no advantage in speed going from RASP1 to RASP1$^{\times}$ programs.

A considerably different extention of RASP's can be obtained by adding <u>associative</u> or <u>content addressed</u> instructions. We illustrate this by listing a few associative operations which are convenient in many computations.

ASS,n ASS,<n>	content of AC is replaced by content of the first register containing n and <n> , respectively, as prefix.
AST,n AST,<n>	stores <AC> in the first register which has n and <n> , respectively, as prefix.

These operations permit the location of desired information without keeping track of its address and one can think of many applications where these operations should speed up the computations. Unfortunately, we have not been able to show for arbitrarily complex computations that we can gain a speed advantage from these associative operations.

What we can show is that for RASP's with sufficiently rich instruction sets the addition of the associative operations can improve the computation speed by no more than the square root.

To make these concepts more precise let a RASP2 with the four additional associative operations ASS,n , ASS,<n> , AST,n , AST,<n> , be denoted by RASP2A.

<u>Theorem</u>: There exists a constant c such that any RASP2A program running in time $T_i(n)$ can be simulated on a RASP2 in time $T_i'(n)$ with

$$c[T_i(n)]^2 \geq T_i'(n)$$

for sufficiently large $T_i(n)$.

<u>Proof</u>: We outline the basic idea of the proof. The memory registers of RASP2 are divided in twenty register blocks. The instructions of any associative program are placed in the odd-numbered memory blocks with the appropriate control statements which change the addresses and transfer control to the block containing the next instruction. The even numbered blocks are used to store the addresses of the registers which the associative program has used. Then, when an associative

operation is encountered a subroutine is entered which, using the list of "used" memory registers, searches through these registers until the desired register is found and executes the prescribed operation.

We observe that the list of "used" registers can grow only at the rate of one register per operation of the RASP2A program. Though these registers can be splattered through the memory, we can search through them in $c(k + \ell)$ operations if the associative program has length ℓ and has performed k operations. Thus for an associative memory which runs for $T_i(n)$ operations our simulation does not require more than

$$T_i'(n) \leq \sum_{p=1}^{T_i(n)} c(p + \ell) = \frac{c}{2} T_i(n)[T_i(n) + 1] + c\ell T_i(n)$$

operations. For increasing $T_i(n)$ and large n we see that

$$T_i'(n) \leq cT_i^2(n) ,$$

as was to be shown.

There are two other RASP extensions whose computation speeds should be investigated and compared to regular and associative RASP's.

The first extention is a list-RASP which has the ability in a single operation to create lists, prefix one list to another, insert a list in another list, delete part of a list, etc. We know little about the capabilities of these machines and have not been able to prove that list-RASP's are faster than conventional RASP's for arbitrarily complex computations.

The second extention is to distributed-logic RASP's. These machines have operations which apply simultaneously to all memory locations. For example, shift every word to the next memory location, if a condition P is satisfied by the content of a register then add it to its predecessor, double the size of the registers satisfying condition P , etc. Again it would be interesting to investigate the computational speed of such devices and compare it to the previous models.

Acknowledgement

The author is glad to acknowledge that this work is strongly influenced by the original discussions of "size arguments" for RASP's with John E. Hopcroft and Peter Wegner as well as by later discussions with Robert L. Constable.

REFERENCES

[1] Elgot, C.C., Robinson, A. "Random-access Stored-program Machines, An Approach to Programming Languages," JACM 11 (1964), 365-399.

[2] Cook, S.A. "Computational Complexity Using Random Access Machines," Course Notes, University of California, Berkeley, 1970.

[3] Hartmanis, J., and Stearns, R.E. "On the Computational Complexity of Algorithms,"Trans. Am. Math. Soc. 117 (1965), 285-306.

[4] Hartmanis, J. "Computational Complexity of One-tape Turing Machine Computations,"JACM, 15 (1968), 325-339.

[5] Hartmanis, J. "Tape Reversal Bounded Turing Machine Computations," JCSS, 2 (1968), 117-135.

[6] Fischer, P.C., Hartmanis, J., Blum, M. "Tape Reversal Complexity Hierarchies," IEEE Conference Record of 1968 Ninth Annual Symposium on Switching and Automata Theory, (1968), 373-382.

[7] Rabin, M.O. "Real Time Computation," Israel Journ. of Math., 1 (1964), 203-211

[8] Hennie, F.C. "One-tape, Off-line Turing Machine Computations," Information and Control, 8 (1965), 553-578.

[9] Hennie, F.C., Stearns, R.E. "Two-tape Simulation of Multi-tape Turing Machines," JACM, 13 (1966), 533-546.

[10] Blum, M. "A Machine-independent Theory of the Complexity of Recursive Functions,"JACM, 14 (1967), 322-336.

[11] Borodin, A. "Complexity Classes of Recursive Functions and the Existence of Complexity Gaps," Conference Record of ACM Symposium on Theory of Computing, 1969, 67-78.

[12] McCreight, E.M., Meyer, A.R. "Classes of Computable Functions Defined by Bounds on Computation: Preliminary Report," Conference Record of ACM Symposium on Theory of Computing, 1969, 79-88.