

Selection of Optimal Computing Platforms through the Suitability Measure^a

Shean T. McMahon
Physical Optics Corporation
20600 Gramercy Pl., Building 100
Torrance, CA 90501-1821
smcmahon@poc.com

Isaac D. Scherson
University of California, Irvine,
Irvine, CA, 90740, USA
isaac@ics.uci.edu

Abstract

Selection of spaceborne computing platforms requires balance among several competing factors. Traditional performance analysis techniques are ill-suited for this purpose due to their overriding concern with runtime. The suitability measure is a new approach that quantifies the match between a computing platform and a program. It analyzes a program at the opcode and control flow levels, and compares this to a machine's capability to support the unique characteristics of the program. In this paper we develop the suitability measure and a series of program analysis methods. Experimental results confirm that machines that provide a better match to the program yield a higher suitability score. We prove that loops provide the only contribution to the suitability value, and also that the number of loop iterations is irrelevant, leading to the conclusion that a single pass through a loop is sufficient to derive a suitability value.

Keywords: Efficiency, opcode, control flow graph, DAG, cycles, irregularity

1. Origin of the suitability measure

One of the most difficult tasks for designers of interstellar and interplanetary probes is selection of the most suitable computing platform. Several factors must be considered, including physical size, weight, power constraints, radiation hardness, cooling issues and computing capability. Providing high performance while satisfying the other constraints often results in tradeoffs such as reductions in performance to meet cooling and power supply restrictions. Because no quantifiable standards currently exist, the choices often rely upon educated guesses. The suitability measure

will provide a method to quantify the match between a program and a machine.

Contemporary performance analysis techniques are concerned primarily with runtime. Instead, the suitability measure quantifies how efficiently a computer executes a program. A machine that runs a program more slowly, but uses its resources more effectively, has a higher suitability than a faster machine that provides poor resource utilization. This makes the measure particularly well suited to spaceborne computing, embedded systems, and other cases where a balance must be struck among a collection of conflicting constraints.

The suitability measure combines opcode-level performance metrics with instruction mix and control flow characteristics collected from static scans of compiled programs. It reveals a machine's efficiency at executing the mean instruction mix seen, over the set of all possible execution paths a program may follow. By approaching the problem from an efficiency perspective, time is factored out of the equations, yielding a time-independent measure.

2.1. Instruction level suitability

Suitability is a composite quantity combining instructional execution efficiency, resource utilization efficiency, and a relative load metric. Each of the terms is unitless. Instructional execution efficiency quantifies a computer's efficiency at executing a single type of instruction. The definition can be relaxed to allow the inclusion of instruction classes or high-level operations such as macros. Its accuracy is inversely proportional to the granularity of the instruction type. It is defined as

$$\eta_i = \frac{\tau_i}{t_i} . \quad (1)$$

a. This work was supported in part by NASA-JPL under Grant 1216595 and by NASA-GSFC under Grant NAG5-9695.

where t_i is the ideal execution time given by the manufacturer's specified latency for the instruction, and t_i is the context-free, actual execution time found using the performance vector [1],[2].

Resource utilization efficiency is the ratio of utilized to available resources in the machine for the instruction in question. Any level of granularity from entire subsystems to individual gates can be considered. Comparisons between machines must use the same resource granularity to be valid. It is defined as

$$U_i = \frac{R_{used_i}}{R_{available_i}} . \quad (2)$$

The relative load measure quantifies the percentage of the program comprising a particular instruction type or class. Collecting this value requires that all possible execution paths in the program be traced. The total number of instructions in the program is given by L_T , while L_i quantifies the frequency of instruction type i . It is defined as

$$Load_{relative_i} = \frac{L_i}{L_T}, \quad L_T = \sum_i L_i . \quad (3)$$

In real-world programs, variations in execution parameters or data will alter the program instruction mix and control flow characteristics. This nondeterministic, irregular behavior complicates collection of the required load metrics. This is overcome by transforming the program control flow (CF) graph into a directed acyclic graph (DAG). Section 3 introduces a series of code structures and program transformations developed for this purpose. Further information on program irregularity can be found in Ref. [3].

Suitability is first defined at the instruction level, and then extended to programs using various combinatory techniques. At the instruction level, it quantifies the efficiency with which a computer can execute a single type or class of instruction. Given an instruction of type i , suitability is defined as

$$S_i = \frac{\eta_i U_i L_i}{L_T} . \quad (4)$$

Partitions form an intermediate step. They combine instructions that share an execution path, and can be viewed as collections of instructions. For partition j , consisting of n instructions of type i , the suitability is defined as

$$S_j = \sum_{i=1}^n S_i = \sum_{i=1}^n \frac{\eta_i U_i L_i}{L_T} . \quad (5)$$

3. Code structures

A crucial step in computing suitability is transformation of the CF graph into a DAG. Code structures form the foundation of this essential step, by providing a vehicle for the extraction of the underlying CF structure and instruction mix of a program. These represent deterministic sequences of opcodes. As such, only a single exit path can exist, and the structure must be traversed in its entirety if encountered. Once the CF structure has been recovered, it is converted into a DAG using a collection of program transformations. From here, a suitability value can be computed. Code structures are similar to the basic blocks created during the compilation process, but differ in their origin, being recovered from compiled, linked, and optimized code.

Code structures fall into two classes: basis forms, specifically the null segment and the unitary sequential segment, and composite forms. Composite forms are constructed by combining the basis forms, and include the maximal sequential segment and the branch and loop structures. The unitary sequential segment is depicted in Figure 1.

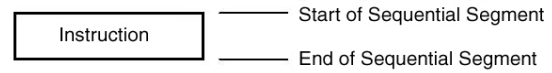


Figure 1. The unitary sequential segment

3.1. Program transformations

In its native state, a program CF graph is both directed and cyclic. Program transformations exploit the behavior of cyclic portions of the program and the suitability definitions, allowing cyclic segments of a program to be transformed into acyclic segments. Several transformations exist, and all are based on the loop proof presented here. Only the loop form is given here, due to space considerations.

3.1.1. The loop proof. Consider a program with four segments as shown in Figure 2. As the number of iterations of the loop segment increases, we expect segments b and c to dominate the program.

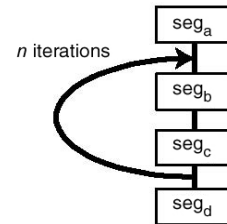


Figure 2. Sequence containing a loop

Let $\gamma = \eta U$ for convenience. The suitability for the sequence in Figure 2 is then given by

$$S = S_a + nS_b + nS_c + S_d$$

$$= \frac{\gamma_a L_a + n(\gamma_b L_b + \gamma_c L_c) + \gamma_d L_d}{L_a + n(L_b + L_c) + L_d} \quad (6)$$

As the iterations n of the loop increase, the suitability S of the sequence converges to the limit of S as n .

$$S = \lim_{n \rightarrow \infty} S$$

$$= \lim_{n \rightarrow \infty} \frac{\gamma_a L_a + n(\gamma_b L_b + \gamma_c L_c) + \gamma_d L_d}{L_a + n(L_b + L_c) + L_d} \quad (7)$$

$$= \frac{n(\gamma_b L_b + \gamma_c L_c)}{n(L_b + L_c)}$$

$$= \frac{\gamma_b L_b + \gamma_c L_c}{L_b + L_c} \quad \text{Q.E.D.}$$

In the limit, the loop segments dominate, allowing the sequential segments to be ignored. This leads to the loop theorem and its corollary.

Loop Theorem: The suitability for a sequence containing a single loop is the instruction count weighted mean U for the loop segment only.

Corollary to the Loop Theorem: **If a sequence contains at least one loop, all non-loop portions of the sequence make no contribution to the suitability value of the sequence.**

Parallel proofs for nested and sequential loops, as well as recursive structures can be developed using a parallel argument. The results of each proof are listed in Table 1. Recall that gamma represents the product of the instructional execution efficiency and the resource utilization efficiency.

Table 1. Loop types and suitability

Type	Suitability
Single loop	Gamma of loop segments
Sequential loops	Instruction count weighted mean gamma of loop segments
Nested loops	Gamma of innermost loop segments
Single recursion	Gamma of recursive segments
Sequential recursions	Instruction count weighted mean gamma of recursive segments
Nested recursions	Gamma of innermost recursive segments

3.2 Partitions

Code structures provided a formalism for disassembling a computer program. This formalism is

necessary for developing the loop proof and its associated machinery, although it is far too restrictive for practical application. Partitions provide a bridge by relaxing the formal requirements of code structures, streamlining the program disassembly process. From here the CF graph can be converted into a DAG, and a suitability value computed. Figure 3 illustrates the structure of a typical partition.

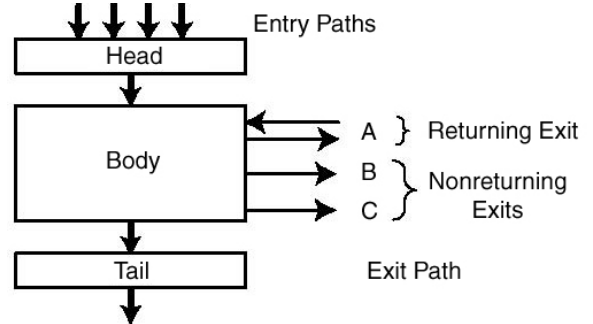


Figure 3. Typical partition structure

Inspection of Figure 3 reveals that partitions allow multiple entry and exit paths in the form of returning and nonreturning exits. Procedure calls are an example of nonreturning exits, while interrupts illustrate returning exits. The determinism constraint of code structures expressly forbids this type of structure. The partitioning procedure is provided in the following pseudocode.

```

Partition_Program
{
  process single line partitions; //interrupts, ...
  build primary partitions; // procedures
  while(cases remain)
  {
    build loop partitions; // loops, recursions,
    process references; // procedure calls
    build minimal partitions;
    remove deadcode;
    return partitioned program;
  }
}

```

Partitioning is an iterative process. It begins with single line opcodes such as interrupts or the IA-32 string-repeat operations [5,6,7]. A partition pointer is substituted for the opcode, and the code itself is placed into a partition. The process continues by extracting code segments until no partitionable code remains. Figure 4 illustrates the partitioning process on a code segment.

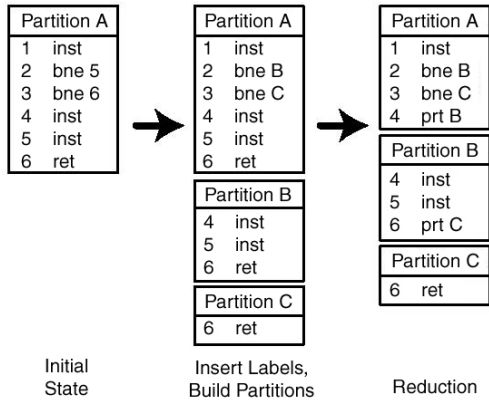


Figure 4. Partitioning of a code segment

A side effect of partitioning is that it reveals dead code, allowing its extraction. Dead code is never accessed during program execution; typically it is inserted during the linking phase of compilation [8,9]. C++ templates have also been shown to be a major culprit in the generation of dead code [10].

3.2.1. Nullification of loop partitions. During partitioning, a series of loop partitions was extracted. Each of these loops contributes to the cyclic nature of the control flow graph, and must be transformed into an acyclic state. The loop theorem and its corollary provide the means to achieve this. The loop theorem reveals that the suitability of a program is entirely derived from the loops, and that only a single pass through each loop is required to extract a suitability value. As such, it is sufficient to simply nullify the loop's internal reference to its entry point to recover the suitability value.

3.2.2. Understanding cyclic paths. The control flow graph of any real program will contain cyclic paths. Both simple and complex cycles occur. A single execution path characterizes simple cycles. In contrast, complex cycles contain multiple paths. Figure 5 illustrates these cases.

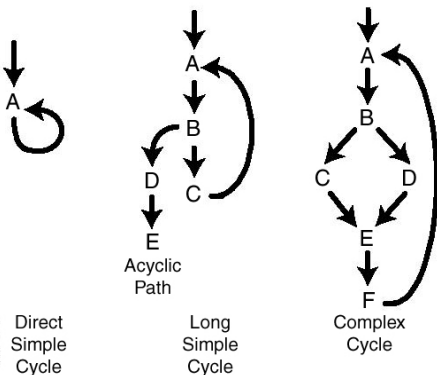


Figure 5. Simple and complex cycles

Extraction of suitability values requires that the cycles be nullified. The effect is to extract a DAG from the program CF graph. For simple cycles, the reference from the tail to the head is nullified, making the cycle acyclic. For complex cycles, no clear path is available, so a mean path must be found. The head of the cycle is located, and each path through the cycle is traced. When no head of the cycle can be located, the principle node must be found. A path trace is then initiated from this node.

3.2.3. Graph reduction and principal node location in complex cycles. The principal node of a complex cycle is the most frequently visited node on the cycle under all possible traversals. Finding the principal node begins by deleting all nodes that do not contribute to the cyclic behavior. Nodes with an indegree or outdegree of zero or one are candidates for deletion under this constraint. All duplicate edges remaining on the graph are then deleted, leading to a reduced graph. Figure 6 illustrates the reduction process.

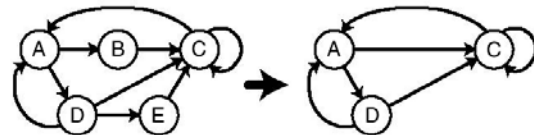


Figure 6. Principal node location reductions

Finally, all possible cycles that exist on the reduced graph are generated. The most frequent is the principal cycle. The head of the principal cycle is the principal node for the entire complex cycle. Next, all possible cycles originating from the principal node are generated. The mean cycle is returned for computation of the suitability value. Figure 7 contains the reduced control flow graph for one of the test programs used in the experiments.

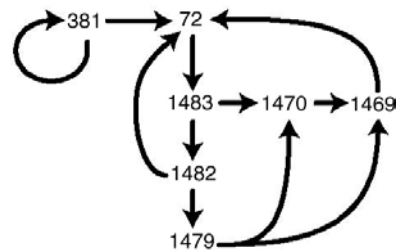


Figure 7. Reduced flow graph for unzipfx.exe

The original complex cycle contained 36 nodes and 48 different paths. The reduction process results in a subgraph with 7 nodes and 11 paths. Inspection

will reveal that the principal cycle is 72 1483 1482 72. Node 72 is the principal node.

4. Recombination rules

Program transformations convert the CF graph into a DAG. Computing the suitability value requires that the gamma and load metrics be known for all possible paths on the DAG. A series of recombination rules are employed to recombine partitions and extract an overall suitability measure for the program/machine system.

When partitions are recombined, the expected gamma and load metrics on each partition are percolated through the DAG to partitions higher in the graph. This process continues until the root partition for the program is reached. The recombination rules summarized in Table 2 govern this process. The results in Table 2 can be proven using construction.

Table 2. Recombination rules

Recombination	Cost	Justification
Seq: S1, S2	Sum of S1, S2	Path composition
Seq: S, L	Loop cost	Loop proof
Branch: S/S	Mean of paths	Path decomposition
Branch: S/L	Min path	Path decomposition
Branch: L/L	Mean of paths	Path decomposition
Loop: single	Loop cost	Loop proof
Loop: Nested	Inner loop cost	Nested loop proof
Loop: Multiple	Mean of loops	Distinct loop proof

Branch S/L: A branch that traverses a sequential or a loop segment

Branch S/S: A branch that traverses one of two sequential segments

Branch L/L: A branch that traverses one of two loop segments

4.1 Computing program suitability

The recombination process replaces each partition reference with the expected gamma and L_i values. These are combined and percolated up through the DAG to the root partition, where the program suitability value can be computed. The suitability is given by the unexpected result

$$S_{prog} = \frac{\gamma_{root} L_{root}}{L_{root}} = \gamma_{root} \quad (8)$$

5. Experimental results

Six test programs were collected from a PC running Microsoft Windows XP Professional, as listed in Table 3.

Table 3. Test programs

Program	Purpose
Agrsmdel.exe	A modem driver
Java.exe	Java application launcher
Javaw.exe	Nonconsole Java application launcher
Mqsvc.exe	Microsoft's message queuing service
Rmi.exe	A remote method invocation application
Unzipsfx.exe	Unzip utility for self-extracting archives

The suitability codes were programmed in Java. Each program was disassembled using Microsoft's Dumpbin binary disassembler. The programs were then partitioned. Operation classes corresponding to those of the MinneSPEC program were chosen [4] for the suitability computation. Table 4 lists them.

Table 4. MinneSPEC operation classes

Instruction Class	Example Opcodes
Integer Computation	add, sub, mul, div, and, ...
Floating Point Comp.	fp_add, fp_sub, fp_sqrt, ...
Load	load
Store	store
Conditional Branch	bne, beq, ...
Unconditional Branch	jmp

The goal of this work was to develop the suitability measure and its associated machinery. Performance vectors would typically be used to gather opcode metrics, but were not a focus of this work. In lieu of generating performance vectors, several representative gamma values for each operation class were chosen. This allowed the partitioning, recombination, and suitability measure procedures to be evaluated.

It was known that the gamma value is in the range of (0, 1]. Thus, choosing a set of values in this range allows for experimental evaluation of a collection of arbitrary machines. It should be noted that it is possible, but uncommon, for the gamma value to exceed unity. If it were to exceed unity, it would merely serve to indicate that the machine was exceptionally well suited to a particular class of operations. In particular, most of the time, it would be operating in a superscalar manner during the execution of this operation.

It was expected that as the efficiency of a machine for common operations increased, its suitability would follow suit. This was found to be the case. Figures 8

and 9 contain plots confirming the expected results for test programs agrsmdel.exe and unzipsfx.exe. The other test programs yielded similar results.

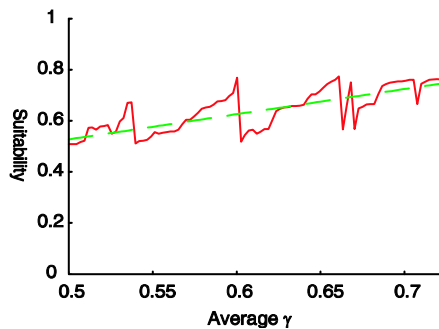


Figure 8. Suitability of agrsmdel.exe

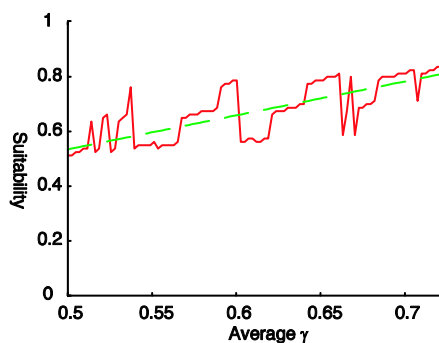


Figure 9. Suitability of mqsvc.exe

6. Future directions and applications

The suitability measure provides a means to evaluate a program and a computer as a system. It combines machine runtime characteristics with program control flow and instruction mix properties, and extracts the efficiency with which the machine can execute the program. Application areas include embedded systems, specialized computing systems, and even general-purpose computers. The unique process required to evaluate suitability admits a method of identifying code bloat, and provides insight into reducing program irregularity. The advantages of this are widespread. One application is providing insight into improved scheduling techniques.

Future directions include investigation of alternate means of opcode metric gathering methods. Performance vectors work, but require significant time for collection. The measure is designed to operate on static scans of programs. Currently, runtime effects are not taken into account. This was not a concern for the initial development effort due to the highly specialized nature of the systems targeted for evaluation. Inclusion of these effects should prove to be straightforward, but is beyond the scope of the current problem.

7. References

- [1] U. Krishnaswamy and I.D. Scherson, "A Framework for Computer Performance Evaluation Using Benchmark Sets," *IEEE Transactions on Computers*, Los Alamitos, CA: IEEE Press, December 2000.
- [2] U. Krishnaswamy, "Computer Performance Evaluation Using Performance Vectors," doctoral thesis, University of California, Irvine, 2000.
- [3] V. Ramakrishnan, "The Convergence of Massively Parallel Processors and Multiprocessors," doctoral thesis, University of California, Irvine, 2000.
- [4] A.J. KleinOsowski and D.J. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research," *Computer Architecture Letters*, vol. 1, June 2002.
- [5] "IA-32 Intel Architecture Software Developer's Manual," vol. 1, Intel Corporation, 2003.
- [6] "IA-32 Intel Architecture Software Developer's Manual," vol. 2, Intel Corporation, 2003.
- [7] "IA-32 Intel Architecture Software Developer's Manual," vol. 3, Intel Corporation, 2003.
- [8] R. Morgan, "Building an Optimizing Compiler," Boston: Digital Press, 1998.
- [9] J.W. Davidson and C.W. Fraser, "Eliminating Redundant Object Code," *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, 1982.
- [10] "Green Hills Software White Paper, Traveling Salesman Demonstration," Green Hills Software, Santa Barbara, CA, <http://www.ghs.com/wp/citiesdemo.html>.