Trace Selection for Compiling Large C Application Programs to Microcode

Pohua P. Chang
Wen-mei W. Hwu

Coordinated Science Laboratory
1101 W. Springfield Ave.
University of Illinois
Urbana, IL 61801

ABSTRACT

Microcode optimization techniques such as code scheduling and resource allocation can benefit significantly by reducing uncertainties in program control flow. A trace selection algorithm with profiling information reduces the uncertainties in program control flow by identifying sequences of frequently invoked basic blocks as traces. These traces are treated as sequential codes for optimization purposes. Optimization based on traces is especially useful when the code size is large and the control structure is complicated enough to defeat hand optimizations. However, most of the experimental results reported to date are based on small benchmarks with simple control structures.

For different trace selection algorithms, we report the distribution of control transfers categorized according to their potential impact on the microcode optimizations. The experimental results are based on ten C application programs which exhibit large code size and complicated control structure. The measured data for each program is accumulated across a large number of input files to ensure the reliability of the result. All experiments are performed automatically using our IMPACT C compiler which contains integrated profiling and analysis tools.

1. Introduction

Code optimization techniques [1] such as register allocation, code compaction, variable renaming, common subexpression elimination, copy propagation, dead code removal, constant folding and strength reduction can perform significantly better by favoring the important execution paths while penalizing the unimportant ones. Trace selection techniques with profiling information identify the important execution paths in terms of frequently invoked sequences of basic blocks.

Trace selection was first proposed by Fisher [5] as a systematic approach to global microcode compaction. Since then, improvements and implementations of optimizations based on trace selection techniques have been reported [9, 10, 11, 12]. These techniques are useful for generating efficient code for application programs which are too large and too complicated to be hand-optimized. However, most of the experimental results reported on using trace selection to assist optimizing large application programs have been based on small benchmarks with simple control structures.

The IMPACT (Illinois Microarchitecture Project using Advanced Compiler Technology) C compiler [8] developed at the University of Illinois employs a set of profiling and analysis tools to guide the trace selection. To collect the profile information for a C program, the compiler automatically inserts function calls into the program to update the profile database. The C programs with probing function calls can then be compiled with standard C compilers (including IMPACT C compiler) to collect profile information on various host machines.

To analyze the profile information for a C program, the compiler automatically associates the profile data with the program control graph and extracts the necessary information for trace selection. The compiler has been stable enough for us to compile large C programs. This allows us to observe the performance of trace selection algorithms on large C application programs over many runs. For different trace selection algorithms, we report the distribution of control transfers categorized according

This research has been supported by the National Science Fundation (NSF) under Grant MIP-8809478, a donation from NCR, the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer laboratory for Aerospace Systems and Software (ICLASS), and the University of Illinois Campus Research Board.

to their potential impacts on the microcode optimizations.

1.1. Structure of the Paper

This paper is divided to six sections. Section 2 describes the automatic profiler and compiler interface. Section 3 provides a brief review and an analysis of the trace scheduling optimization. Section 4 shows three trace selection functions. Section 5 outlines the experiment and presents the experimental result. In section 6, we offer some concluding remarks.

2. Weighted Program Control Graph

In this section, we first introduce the notion of weighted program control graph. Then, we briefly describe how the automatic profiler and the C compiler collaborate to construct the weighted program control graph.

2.1. Program Representation

In our C compiler, a program is represented by a weighted program control graph. Weighted program control graph is a directed graph where every node is a basic block, and every arc is a branch path between two basic blocks. The weight of a node is the average execution count of the corresponding basic block for a single run of the program. The weight of an arc is the average execution count of the corresponding branch path for a single run of the program.

For example, given that the program consists of three basic blocks A, B, and C, where A has an average execution count of 100, B has an average execution count of 90, and C has an average execution count of 10 in a single run. The last instruction of A is a conditional branch to either B or C depending on some branch condition. The arc A->B has a weight of 90 and the arc A->C has a weight of 10. Then one can conclude that A is the only immediate predecessor of B and C, because *the sum of the weights of all incoming arcs = the sum of the weights of all outgoing arcs = the node weight.*

The arc A->B is said to be an *outgoing arc* of node A, and is an *incoming arc* of node B. From the opposite perspective, node A is the *source* and node B is the *destination* of the arc A->B. A node may have several incoming and outgoing arcs.

2.2. Control Graph Construction

The IMPACT C compiler applies constant folding, dead code removal, and jump optimization to the program control graph to derive a transformed control graph with fewer and larger basic blocks. Then it inserts additional code in all basic blocks to collect node and arc weights dynamically.

Upon completion of a single program run, a profiler routine is automatically activated to store the profiled result into a database. The profiler logs the node and arc weights, and the number of times the program has been profiled. For each run, the profiler updates the program profile information according to *[W.permanent = W.permanent*N/(N+1) + W.single.run/(N+1); N=N+1]* where N is the number of time the program has been profiled, *W.permanent* is the accumulated weight, and *W.single.run* is the weight calculated in the last run.

The profiler provides functions which can be called by the compiler to obtain the profile information. The compiler reads in the profile information and assigns node and arc weights of the program control graph.

The resultant weighted control graph is the program representation used to study trace selection in the next section.

3. Trace Scheduling

We refer readers who are unfamiliar with trace scheduling to the original paper by Fisher [5]. Trace scheduling consists of three major functions : *trace selection*, *local compaction*, and *bookkeep*. First, the trace selection function selects the most likely to be executed program path. Then, local compaction is applied to schedule the trace. And finally, the bookkeep function inserts patch code at the *split* and *rejoin* points to preserve correctness. The three functions are described in great detail in Ellis's thesis [11].

Trace scheduling permits the patch code created during the bookkeep phase of a trace to be selected and compacted as part of later traces. However, we do not allow the additional basic blocks generated by the bookkeep function, unless they can be absorbed by jump optimization, to be considered when forming later traces. This requirement allows us to apply trace selection independently of the local compaction and bookkeep functions.

Code motion moves critical instructions on the program critical paths up to the earliest point that they can be executed. The usefulness of the code motion and the cost of the bookkeeping on the total program execution time depends on the program structure and also on the underlying microarchitecture. For example, code motion applied to a section of a program with large fine-grain parallelism will tend to do well due to the large code movement freedom. In a pipelined processor, code motion allows the execution of multi-cycle operations to overlap with the issuing and execution of less critical operations when there is no data dependence. Similarly in a processor capable of issuing multiple instructions per cycle, code motion reduces execution time by compacking operations into fewer instructions.

Trace scheduling guides global code motion by favoring most frequently executed program paths. Therefore the goal of the trace selection function is to identify when forming longer traces are desirable and how all basic blocks should be partitioned to various traces. It would be grossly complicated for the trace selection function to deal with micro-architecture dependent factors such as degree of hardware parallelism. Disregarding the hardware limitations, the trace selection function try to form the longest possible traces, limited only by program dependent factors.

The question is what program dependent factors must the trace selection function consider. The program control flow, local program parallelism, and the code mobility as determined by data-flow analysis can all be implemented in the trace selector. The program flow analysis, either by loop analysis or dynamic profiling, allows the trace selector to form traces by grouping series of basic blocks which tend to execute together. The local program parallelism and code mobility analysis tells the trace selector when trace expansion should be stopped due to limited code movement freedom. However, the complexity of the analysis, although required in later phases of compilation, hinders the development of a clean selection function. It is best to use only the control flow information and to construct the longest traces.

Our IMPACT C compiler allows automatic profiling and provides accurate execution weights for all control graph nodes and arcs. The problem now is how to form traces in such a way that the in-trace transition is maximized and the off-trace transition is minimized. Off-trace transitions can be finer partitioned to five different types. Together with in-trace transition, there are a total of six transisiton types (T1-T6).

T1 connects the last node of a trace to the start node of a different trace.
T2 connects the last node of a trace to a middle node of a trace.
T3 connects a middle node of a trace to the start node of a trace.
T4 connects two middle nodes.
T5 connects two nodes within a trace.
T6 connects the last node of a trace to the start node of the same trace.

Code motion is permited only for T5 connections. T2 transition requires bookeeping at the rejoin location. T3 transition requires bookeeping at the branch location. T4 connections require bookeeping at both the branch and the rejoin locations. T2, T3, and T4 thus may execute longer than the same code without applying trace scheduling. Global code motion is not allowed across T1 and T6 connections, and therefore obtains no speedup over local code compaction.

Let %a, %b, %c, %d, %e and %f denote the percentage of T1, T2, T3, T4, T5 and T6 transitions respectively, in a typical program run. The goal of the trace selector is to maximize %e and to minimize %b, %c, and %d.

The various percentages allow us to compare different trace selection functions. A trace selection function is better than others if it generates higher %e and lower %b, %c, and %d, for a given control graph.

4. Trace Selection

4.1. General Selection Function

In his trace scheduling paper [5], Fisher presented the following trace selection algorithm with node weights as the selection criteria. Later, Ellis in his thesis [11] implemented the same general trace selection algorithm but use arc weights as the selection criteria.

```
algorithm trace_selection
 mark all nodes unvisited;
 while (there are unvisited nodes)
  /* select a seed */
  seed = the node with the largest execution
      count among all unvisited nodes;
  mark seed visited;
  /* grow the trace forward */
  current = seed;
  loop
   s = best_sucessor_of(current);
   if (s==0) exit loop;
   add s to the trace;
   mark s visited;
   current = s;
  /* grow the trace backward */
  current = seed;
  loop
   s = best_predecessor_of(current);
   if (s==0) exit loop;
   add s to the trace;
   mark s visited;
   current = s;
  /* compaction and bookkeep */
  trace_compaction;
  book_keep;
```

Since we do not consider the additional basic blocks generated by the *book_keep* function in the trace selection process, the *trace_compaction* and the *book_keep* functions are not included in the above algorithm.

To ensure that loop headers become the leading nodes of traces, in growing trace forward and backward, crossing loop back-edges is prohibited.

4.2. Selection According to Node Weight

Node weight is the execution count of a basic block. This number can either be estimated statically by loop analysis [1], or dynamically profiled by an automatic profiler. In this paper, all weights used in the trace selection functions are strictly derived from the average program profile accumulated over many runs.

```
best_successor_of(node)
  n = Of all immediate successors of node,
    n has the highest execution count;
  if (n is visited) return 0;
  return n;
best_predecessor_of(node)
  n = Of all immediate predecessors of node,
    n has the highest execution count;
  if (n is visited) return 0;
  return n;
```

4.3. Selection According to Arc Weight

Each node (basic block) of the control graph can have several incoming and outgoing arcs. Each arc represents a possible branch path connecting two nodes. Trace scheduling yields some performance gain when the program flows through an arc within a trace, and suffers when an off-trace is taken. Hence, arc weight is a better selection criterian than node weight.

```
best_successor_of(node)
  e = Of all edges leaving node, e has the
    highest execution count (highest probability);
  n = the destination of e;
  if (n is visited) return 0;
  return n;
best_predecessor_of(node)
  e = Of all edges entering node, e has the
    highest execution count (highest probability);
  n = the source of e;
  if (n is visited) return 0;
  return n;
```

4.4. Selection with Minimum Arc Probability Requirement

Some nodes have many incoming and outgoing arcs. If there is not a single arc which dominates all others, the performance gain that can be extracted by including the most likely to be taken arc by a trace will be overshadowed by the combined off-trace cost of all other arcs. In such instances, it is better to stop the trace expansion. To detect such cases, a minimum arc probability requirement is added to the selection function.

The probability that an outgoing arc Ai will be taken, given that the program control is already at node Nj which is the source of Ai, is simply [arc_weight(Ai) / node_weight(Nj)]. The probability a node Na is reached through an arc Ab is [arc_weight(Ab) / node_weight(Na)]. In section 5, we measure the performance of this selection heuristic with several MIN_PROB values.

```
best_successor_of(node)
  e = Of all edges leaving node, e has the
    highest execution count (highest probability);
  if (probability(e) <= MIN_PROB) return 0;
  n = the destination of e;
  if (n is visited) return 0;
  return n;
best_predecessor_of(node)
  e = Of all edges entering node, e has the
    highest execution count (highest probability);
  if (probability(e) <= MIN_PROB) return 0;
  n = the source of e;
  if (n is visited) return 0;
  return n;
probability(e)
  s = source of e;
  d = destination of e;
  return min((weight(e)/weight(s)),
          (weight(e)/weight(d)));
```

5. Experiments

5.1. Procedure

The compiler compiles and profiles the benchmark programs by inserting extra code to record the execution count of basic blocks and branch paths. The compiled programs are installed and tested with many inputs. For each run, the profiler updates the accumulated average execution count of basic blocks and branch paths for a typical run of the program. With the profile information, the compiler constructs the weighted control graph. Then, trace selection is applied to the weighted control graph, and the percentage of the six connection types (%a %b %c %d %e %f) are measured.

5.2. The Benchmark

Ten programs from several application domains are chosen mainly because of their popularity and substantial program size. Each of the ten programs is run at least ten times with realistic inputs. We have also made a special effort to exercise nearly all program options.

In table 1, the *name* column lists the program name, the *line* column shows the number of non-empty lines of C code after preprocessing in each of the benchmarks. The *run* column indicates the number of runs under profiler monitoring.

5.3. Percentage of Transaction Types

We report the percentage of each of the six transition types executed in a typical run of the benchmark program. The *loop* column in the following tables is the average number of basic blocks in a executed inner loop. The *trace* column is the average number of basic blocks of all traces executed. Table 2 corresponds to the selection according to node weight function. Table 3 corresponds to the selection according to arc weight function. Table 4 to 7 demonstrates the effect of imposing additional minimum branch probability requirement.

5.4. Discussion of Result

As we have expected, arc weight is a better selection criterian than node weight. The additional minimum branch probability requirement furthur reduces the off-trace cost. As the minimum branch probability requirement increases, %b, %c, and %d percentages decline slightly. However as the minimum requirement rises, fewer and smaller traces are formed, leading to low percentage of in-trace transitions.

In any case, the in-trace transition (%e) is several times larger than the off-trace transitions (%b, %c, %d). This essentially tells us that even a small improvement in in-trace code movement can compensate for much larger bookkeep cost.

The off-trace transitions (%b, %c, %d) are low, because benchmark programs have predictable branch behavior. The profile information shows that, on the average, the branch direction of more than 90% of all branch instructions executed can be correctly predicted statically. Excluding function calls and returns, the average control flow predictability, including all conditional, unconditional and multi-way branchs, for the benchmarks is summerized in table 8.

table 8 : Control Flow Predictability.

cpp	.8803	*eqn*	.9561
espresso	.8095	*grep*	.9632
more	.9764	*mpla*	.9226
nroff	.9770	*pic*	.9553
tbl	.9658	*wc*	.9250

A few of the benchmark programs show substantial inner loop back-edge transitions (%f). Loop unrolling can be applied to exploit program parallelism across loop iterations. When N copies of a loop exist, the loop back-edge of the first (N-1) instances can be transformed into normal connection between two distinct nodes. These (N-1) connections between different iterations of the loop can be selected for trace expansion. Since many iterations are usually taken before the program control leaves the loop, the expanded loop structure will form a long trace covering the most important path of all unrolled instances of the loop.

For several benchmarks, the number of function calls are substantial, more than one function call per every six basic blocks executed. The program *tbl* shows the highest function call frequency, about one function call for every two basic blocks executed. The profile result shows that the most frequently executed function in *tbl* consists of only one basic block. Similarly in the other programs, the most frequently executed functions tend to be small, and can be easily in-line expanded. Since function in-line expansion not only gives larger traces, but also eliminates register saving and restoring around the function boundaries, the potential gain seems to be more substantial than loop unrolling.

Of all traces actually executed, the average trace size is about three to four basic blocks for various selection functions. The relatively small size is due to control uncertainties and small function body. One can expect some increase in trace length after function in-line expansion.

An inner loop as seen by the IMPACT C compiler is a trace whose last node branches back to the trace header. The average size of all inner loops executed is about three basic blocks. In another word, one can expect two conditional branchs in inner loops. Therefore, loop unrolling and software pipelining techniques for large integer programs must cope with at least two conditional branchs in inner loops.

5.5. Bookkeeping Cost

Since the percentage of off-trace transition (%b, %c, %d) is much smaller than in-trace transition (%e), trace scheduling can tolerate large off-trace cost. In section three, we have stated that the new basic blocks generated by the bookkeep function will not be considered in forming later traces. The performance penalty of that decision in terms of execution time is small.

6. Conclusion

Using profiling data in our trace selection algorithm, we have reduced some control uncertainties. Furthermore, our experiments with various trace selection functions have shown that trace scheduling can guide global code motion effectively with very little off-trace penalty. The percentage of off-trace transitions (%b, %c, %d) can be reduced by increasing the minimum branch probability requirement. For some of the benchmark programs we have tested, function in-line expansion and loop unrolling should be exploited to obtain additional performance.

Acknowledgements

The authors would like to acknowledge Sadun Anik, Nancy Warter, Thomas Conte, and the other members of the Computer System Group for their invaluable comments and suggestions.

Reference

[1] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986.

[2] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure and M. Wolfe, "Dependence Graphs and Compiler Optimizations," Proceedings of the 8th ACM *Symposium on Principles of Programming Languages*, Jan., 1981.

[3] Ron Cytron and Jeanne Ferrante, "The Value of Renaming for Parallelism Detection and Storage Allocation," Proceedings of the 1987 *International Conference on Parallel Processing*, Aug., 1987.

[4] Mario Tokoro, Eiji Tamura and Takashi Takizuka, "Optimization of Microprograms," IEEE Transactions on Computers, vol. c-30, no.7, July, 1981.

[5] Joseph A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," IEEE Transactions on Computers, vol. c-30, no.7, July, 1981.

[6] John Hennessy and Thomas Gross, "Postpass Code Optimization of Pipeline Constraints," ACM Transactions on Programming Languages and Systems, vol. 5, no.3, July, 1983.

[7] B. Su, S. Ding, and J. Xia, "URPR - An Extension of URCR for Software Pipelining," Proceedings of the 19th *Microprogramming Workshop*, New York, NY, Dec., 1986.

[8] Wen-mei W. Hwu and Pohua P. Chang, "Exploiting Parallel Microprocessor Microarchitectures with a Compiler Code Generator," The 15th *Annual International Symposium on Computer Architecture*, Honolulu, Hawaii, May, 1988.

[9] J.L. Linn, "SRDAG Compaction: A Generalization of Trace Scheduling to Increase the Use of Global Context Information," Proceedings of the 16th *Microprogramming Workshop*, Downingtown, PA., Oct., 1983.

[10] B. Su, S. Ding, and L. Jin, "An Improvement of Trace Scheduling for Global Microcode Compaction," Proceedings of the 17th *Microprogramming Workshop*, New Orleans, LA., Nov., 1984.

[11] J.R. Ellis, Bulldog: A Compiler for VLIW Architectures. The MIT Press, 1985, PhD thesis, Yale, 1984.

[12] Michael A. Howland, Robert A. Mueller and Philip H. Sweany, "Trace Scheduling Optimization in a Retargetable Microcode Compiler," Proceedings of the 20th *International Microprogramming Workshop*, Colorado Springs, Dec., 1987.

table 1 : Benchmark Set.

name	*line*	*run*	*description*
cpp	3355	34	GNU C preprocessor
eqn	3775	10	typeset mathematics for nroff/ditroff
espresso	10405	18	boolean minimization
grep	447	10	pattern search
more	1644	10	browse through a text file
mpla	1134	18	technology independent PLA generator
nroff	10263	10	format documents for display
pic	7916	20	format pictures for nroff/ditroff
tbl	3403	14	format tables for nroff/ditroff
wc	116	10	word count program

table 2 : Selection According to Node Weight.

	%a	%b	%c	%d	%e	%f	loop	trace
cpp	.1387	.0350	.1054	.0106	.3764	.3339	1.78	1.84
eqn	.0416	.1740	.1796	.0000	.5631	.0419	3.95	2.58
espresso	.2632	.0813	.1285	.0781	.2941	.1547	1.97	1.88
grep	.2743	.0984	.1077	.0038	.4379	.0780	3.01	2.17
more	.0956	.1366	.1380	.0095	.5964	.0240	4.74	3.83
mpla	.1090	.0611	.0848	.1276	.5309	.0866	3.88	2.78
nroff	.0246	.0981	.1062	.0120	.7159	.0414	5.39	3.74
pic	.0202	.1009	.1085	.0203	.7104	.0386	2.02	3.57
tbl	.0343	.0816	.0909	.0050	.7023	.0859	1.90	2.45
wc	.0943	.1093	.1369	.0000	.5734	.0860	6.00	3.25

table 3 : Selection According to Arc Weight.

	%a	%b	%c	%d	%e	%f	loop	trace
cpp	.1256	.0098	.0804	.0203	.4299	.3339	1.82	1.97
eqn	.1969	.0103	.0219	.0215	.7314	.0181	1.33	3.07
espresso	.1490	.0571	.0968	.1862	.4025	.1083	2.10	2.16
grep	.1775	.0206	.0289	.0087	.6799	.0845	4.93	3.42
more	.2010	.0160	.0213	.0073	.7514	.0031	2.95	4.41
mpla	.1232	.0468	.0737	.1276	.5421	.0865	3.88	2.77
nroff	.0506	.0079	.0169	.0184	.8711	.0356	6.66	5.11
pic	.0942	.0147	.0411	.0133	.7928	.0439	5.60	3.94
tbl	.0649	.0077	.0176	.0153	.8128	.0814	1.52	2.73
wc	.0703	.0038	.0278	.0241	.7880	.0860	7.00	5.73

table 4 : Minimum Branch Probability = 60%.								
	%a	%b	%c	%d	%e	%f	loop	trace
cpp	.3344	.0098	.0212	.0200	.3553	.2591	1.74	1.64
eqn	.2171	.0074	.0186	.0090	.7298	.0181	1.40	2.94
espresso	.2304	.0465	.0764	.1615	.3667	.1185	1.94	1.82
grep	.1918	.0172	.0240	.0042	.6758	.0866	4.89	3.31
more	.2014	.0160	.0212	.0073	.7512	.0031	2.95	4.40
mpla	.2897	.0087	.0231	.1270	.4938	.0577	3.19	2.14
nroff	.0568	.0073	.0144	.0178	.8680	.0356	6.64	5.00
pic	.1296	.0134	.0312	.0102	.7799	.0358	1.99	3.19
tbl	.0735	.0071	.0145	.0155	.8080	.0814	1.48	2.65
wc	.0703	.0038	.0278	.0241	.7880	.0860	7.00	5.73

table 5 : Minimum Branch Probability = 70%.								
	%a	%b	%c	%d	%e	%f	loop	trace
cpp	.3582	.0090	.0146	.0181	.3409	.2590	1.74	1.58
eqn	.2372	.0051	.0144	.0082	.7186	.0166	1.31	2.65
espresso	.5663	.0145	.0217	.0870	.2017	.1089	1.88	1.57
grep	.0198	.0157	.0240	.0002	.6746	.0866	4.89	3.24
more	.2016	.0159	.0211	.0072	.7512	.0031	2.99	4.40
mpla	.2899	.0087	.0231	.1270	.4937	.0577	3.19	2.14
nroff	.0585	.0071	.0140	.0177	.8670	.0356	6.64	4.97
pic	.1498	.0106	.0256	.0102	.7691	.0349	1.85	2.83
tbl	.0906	.0069	.0105	.0134	.7972	.0814	1.48	2.56
wc	.0703	.0038	.0278	.0241	.7880	.0860	7.00	5.73

table 6 : Minimum Branch Probability = 80%.								
	%a	%b	%c	%d	%e	%f	loop	trace
cpp	.4053	.0055	.0111	.0144	.3103	.2532	1.67	1.49
eqn	.2686	.0013	.0086	.0077	.6972	.0166	1.31	2.44
espresso	.6748	.0076	.0081	.0518	.1569	.1008	1.74	1.43
grep	.1990	.0157	.0240	.0002	.6745	.0866	4.89	3.24
more	.2021	.0158	.0210	.0074	.7507	.0031	2.95	4.39
mpla	.3286	.0084	.0138	.1270	.4646	.0577	3.19	1.97
nroff	.0862	.0046	.0128	.0711	.7942	.0311	2.87	4.22
pic	.2109	.0028	.0124	.0174	.7217	.0350	1.90	2.27
tbl	.1128	.0065	.0095	.0110	.7788	.0813	1.48	2.46
wc	.0703	.0038	.0278	.0241	.7880	.0860	7.00	5.73

table 7 : Minimum Branch Probability = 90%.								
	%a	%b	%c	%d	%e	%f	loop	trace
cpp	.4468	.0044	.0097	.0082	.2881	.2427	1.61	1.43
eqn	.2827	.0009	.0069	.0077	.6866	.0154	1.22	2.36
espresso	.7661	.0007	.0019	.0065	.1417	.0831	1.34	1.20
grep	.2952	.0002	.0083	.0000	.6097	.0866	4.89	2.58
more	.2912	.0014	.0063	.0073	.6908	.0031	2.95	3.34
mpla	.3902	.0063	.0081	.1270	.4270	.0413	3.48	1.81
nroff	.1762	.0013	.0037	.0696	.7295	.0197	2.63	3.28
pic	.3254	.0012	.0018	.0054	.6491	.0171	1.72	1.98
tbl	.1265	.0053	.0083	.0114	.7681	.0804	1.46	2.42
wc	.5798	.0002	.0002	.0000	.4197	.0000	0.00	1.72