U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology

NISTIR 5131

Computer Systems Laboratory

Synthetic-Perturbation Tuning of MIMD Programs

Gordon Lyon
Robert Snelick
Raghu Kacker

U. S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Gaithersburg, MD 20899

February 1993

CMRF

COMPUTER MEASUREMENT
RESEARCH FACILITY
FOR HIGH PERFORMANCE
PARALLEL COMPUTATION

Partially sponsored by the
Defense Advanced Research Projects Agency

NISTIR 5131

Synthetic-Perturbation Tuning of MIMD Programs

Gordon Lyon, Div. 875
Robert Snelick, Div. 875
Raghu Kacker, Div. 882

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Gaithersburg, MD 20899

Partially sponsored by the Defense Advanced Research Projects Agency.

This expanded version supersedes NISTIR 4859 of June, 1992.

February 1993

TABLE OF CONTENTS

Page

1. A Problem with MIMD 2

2. Synthetic-Perturbation Tuning 2

    2.1 Description of Technique 3

        2.1.1 Treatment Opportunities 4
        2.1.2 In Practice 6
        2.1.3 The Screening Model 6

3. SPT on Shared and Distributed-Memory Systems 8

    3.1 Example 1-SM: Tuning a Parallel Quicksort 8

        3.1.1 An Approach via the Tool *gprof* 8
        3.1.2 New Approach 10
        3.1.3 Shared-Memory Results 13

    3.2 Example 2-DM: Ring-Connected Nodes 14

        3.2.1 Optimization Rather than Screening 16
        3.2.2 Distributed-Memory Results 18

4. Different Delays, Distinct Systems 18

    4.1 When Delay Choice Matters Less: Example 3-SM 19

    4.2 Higher Unpredictability: Example 3-DM 21

5. Conclusions 22

    5.1 Further Directions 23

    5.2 Summary 24

6. References 25

Synthetic-Perturbation Tuning of MIMD Programs

Gordon Lyon, Div. 875
Robert Snelick, Div. 875
Raghu Kacker, Div. 882

Synthetic-perturbation tuning--SPT--is a novel technique for assaying and improving the performance of programs on MIMD systems. Conceptually, SPT brings the powerful, mathematical perspective of statistically designed experiments (DEX) to the interdependent, sometimes refractory aspects of MIMD program tuning. Practically, SPT provides a needed reconfiguration mechanism via synthetic delays for what otherwise would be *ad hoc*, hand-tailored program setups for DEX. Overall, the technique identifies bottlenecks in programs directly as quantitative effects upon response time. SPT works on programs for both shared and distributed-memory and it scales well with increasing system size.

Key words: code perturbation; designed experiments; factorial designs; MIMD; parallel programming; performance improvement; synthetic delays.

No recommendation or endorsement, express or otherwise, is given by the National Institute of Standards and Technology or any sponsor for any illustrative commercial items in the text. Partially sponsored by the Defense Advanced Research Projects Agency, ARPA Task No. 7066. Part of the text appears in the Proceedings of the Sixth International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, 16-18 September, 1992, Edinburgh. These are published as **Performance Tools '92**, R. Pooley and J. Hillston (eds.), Edinburgh University Press Ltd.

1. A Problem with MIMD

MIMD programs are difficult to code well and to improve. The asynchronous, interdependent nature of concurrent events makes this so. For unlike serially-executed computer programs, MIMD parallel code has performance interactions that are not readily predictable. Much depends upon communication, including synchronization, among the parallel components. For example, communication-induced idle time exerts an important but elusive influence (illustrations follow). The conventional approach to MIMD tuning offers users an instrumentation that captures important performance details. Unfortunately, it is clear that scalable parallel systems, once instrumented, can generate an avalanche of internal detail. Most of this information should be kept hidden from users. Yet, to improve their codes, MIMD programmers must identify program bottlenecks. Conversely, programmers should know which sections of code can be expanded with only minor performance penalties. Although isolated changes in MIMD code can be monitored for overall performance effects, a typical parallel program will have a large number of potential improvements. This number will defeat any naive attempt at a comprehensive evaluation.

2. Synthetic-Perturbation Tuning

An alternate approach suggests treating the MIMD program as a "black box" that has input parameters and an output. This empirical paradigm is common in the control of highly complex industrial processes such as chemical manufacture. MIMD programs display a similarly complex character. Delay routines inserted into a program provide a convenient set of input test parameters that may otherwise be lacking. The delays fit easily into code, since they are synthetic quantities divorced from normal computational states. This is synthetic-perturbation tuning, *SPT* [1].

Response times are measured for runs performed with systematic patterns of the delays. A macro-level response model then predicts the program's sensitivity to each of the delays. Internal details of the program are largely ignored, a philosophy that contrasts sharply with conventional tuning approaches. SPT rests heavily upon the design of experiments (DEX), a modern branch of statistical theory well-suited for studying complex systems [2]. DEX provides *quantitative* estimates of the effects of SPT's synthetic bottlenecks. This establishes quickly and accurately which locations matter most for improvement.

2.1 Description of Technique

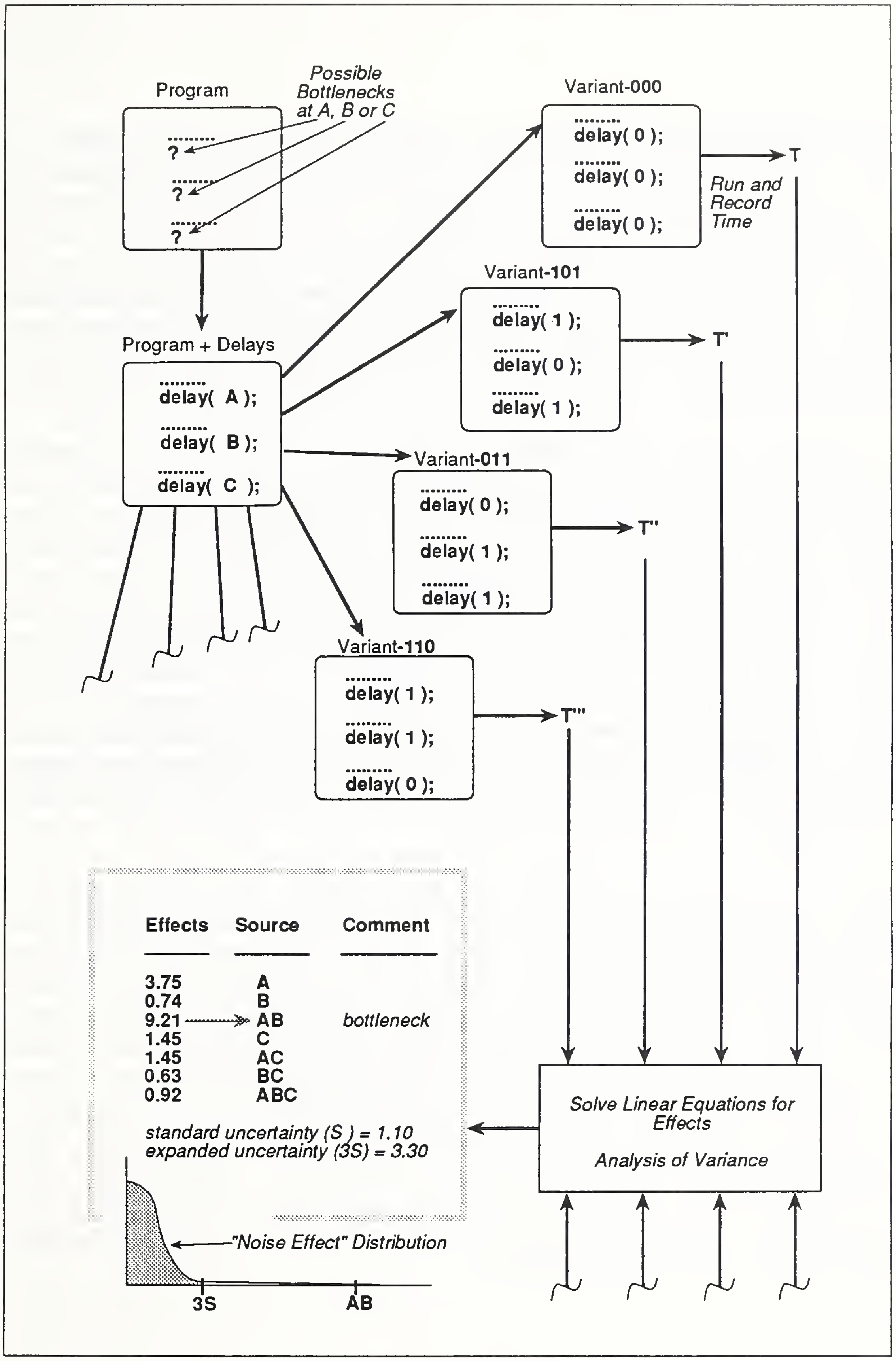
Synthetic-perturbation tuning involves (i) a specimen program, *P*, (ii) a response to be improved, *e.g.*, runtime of *P*, (iii) mathematical methods from the statistical design of experiments, and (iv) standard, synthetic methods of delay (or similar perturbations) that are inserted into program *P*. A starting set of test locations is refined iteratively as sensitivities become known. SPT has the following steps:

1. Select code locations in program *P* to be tested, via synthetic delays, for their effect upon performance. These *p* locations are called *factors*. Each delay is a *factor treatment*.
2. Insert standard perturbations in treatment patterns over the factors. This will generate numerous treated versions of the program. Order these randomly.
3. Run each treated version of the program and record its *response*. This is a *trial*. Trial *replications* should use a fresh randomization order.
4. Analyze the measured responses with a *linear response model*. This typically employs *ANOVA*, the analysis of variance. In some circumstances, the linear *response surface* result is inappropriate, and more informal methods are used (an example follows in the text).
5. Assess step 4 and select those *effects* that appear significant. These effects correspond to single factors and *multiple–factor interactions* that are sensitive to delay treatment. (Figure 1 has an example.) Unimportant factors are dropped and new factors added.
6. Repeat steps 1-5 as needed. Analysis proceeds in cycles of refinement.
7. Improve sensitive factors of program *P*. (The nature of this improvement is beyond the purview of SPT.) If desired, return to step 1.

2.1.1 Treatment Opportunities. There are numerous ways to treat program *P*. In perhaps the easiest and most useful case, delays are inserted into *source-level control paths* prior to compilation. All examples discussed here use this form of perturbation. However, the chosen treatment does not have to be a time delay. Other treatments include perturbations that (i) lock and unlock some important variable while doing nothing substantive to it, (ii) send dummy messages, or (iii) acquire temporary buffer space. Another possibility for source code treatment is to insert delays along *data paths*. An illustration for such a delay treatment may be useful. Let integer variable *A* be a factor. Replace references to *A* by invocations of an integer function *ifu*, so that *ifu*(A,x) returns the value of *A* after a delay of *x* units. All invocations of *ifu*(A,x) constitute treatment for one factor, the reference of *A*. Aliasing must be examined carefully in this context, since pointers to *A* may elude a static source-level treatment.

Below source-level treatments lie system-level approaches. Here, the available resources will determine what can be done. Processors that can trap specific location references (*e.g.*, with *watchpoint registers*) can always delay anonymous pointer references. On other systems, trap-on-address may be difficult. The case for control delays is easier. Perturbations can be patched into loadable code (after compilation and linkage) by the familiar technique of jumping out of the original code to a perturbation table and then jumping back.

While other possibilities exist, the principle remains: The extra perturbation code neither omits nor reorders any code of an original thread. Nor does the perturbation depend upon the computational nature of the original code; such would be the case if treatment meant code had to be tediously rewritten to be slower or faster while giving the same results. Fortunately, such rewriting is unnecessary. For example, a synthetic delay function does not use global names or values from the original code. Because computational soundness of a treated program always remains intact, perturbations can be inserted, tested and removed swiftly. The perturbation used at all points is standard; it is not tailored *ad hoc* for each code context. This fact is pivotal to practical applications of SPT, including the automatic generation of trials.



Effects	Source	Comment
3.75	A	
0.74	B	
9.21	AB	bottleneck
1.45	C	
1.45	AC	
0.63	BC	
0.92	ABC	

standard uncertainty (S) = 1.10
expanded uncertainty (3S) = 3.30

Figure 1: Example Flow of SPT (Full Factorial) Analysis

2.1.2 In Practice. Each inserted delay treatment simulates added instructions. This delay function is not complicated. In the distributed-memory experiment that follows (Example 2-DM), delays are generated by a recursive function, *delay* (X), embedded in a compilation directive. $X=0$ generates no executable code. For $X>0$, *delay* (X) performs X invocations, X floating-point multiplies, and X returns. The value of X is determined by the nature of the program P, the system, and the fineness of detail being investigated. $X=10$ is typical. Values of X can vary from factor to factor if this is appropriate, but except for example 2-DM (follows), a uniform treatment setting is used throughout a program.

Patterns of delays are determined once p program test factors (locations) have been chosen. SPT starts by treating each factor at two delay settings, one of which is zero delay. Given the p factors, an exhaustive experiment will involve 2^p patterns. This is a *full factorial* design [2]. *Partial factorial* experiments, designated 2^{p-k}, generate far fewer patterns but have less experimental resolution of interaction effects. Partial factorial experiments are generally quite adequate. SPT provides knowledge about factors indirectly by examining their combined effects over numerous trials. DEX analysis works backwards from response times and the pattern of treatment associated with each time. While all examples here use runtime as a response, maximum consumed space is another possibility and transactions per minute, a third. More than one response can be measured for each trial; each distinct response corresponds to a separate experiment. Thus, each experiment requires multiple trials but these trials can serve other experiments.

2.1.3 The Screening Model. SPT incorporates a screening method. More so than methods of regression, optimization or comparison, a statistical screening focuses upon identifying which among tested factors matter most. The analysis rests upon a simple DEX response surface model that assumes linearity in all variables. Factor *treatment levels* therefore need assume only two values, here represented after *rescaling* as $(-1,+1)$. Abbreviations minus $(-)$ and plus $(+)$ are used in sequel for -1 and $+1$ settings of treatment. Interaction settings are determined from factor treatment settings, so that $X_A = -$ and $X_B = +$ imply the interaction $X_{AB} = (-)\times(+) = -$. These binary settings are generally adequate, although exceptions are explored later. Still, circumstances suggest first investing only a minimum of effort in each screening, since program tuning will change the very code under study (step 7). The utility of a particular response surface model is quite short-lived.

Suppose there are program factors A, B and C with corresponding treatment variables of X_A, X_B and X_C. The response surface model is:

$$R = \mu + ½ [\beta_A X_A + \beta_B X_B + \beta_{AB} X_{AB} + \beta_C X_C + \beta_{AC} X_{AC} + \beta_{BC} X_{BC} + \beta_{ABC} X_{ABC}] \quad (1)$$

Multiplier ½ arises because the domains of treatment variables $\{X_j\}$ span $[-1, +1]$, a distance of two. The 2^3 unknowns to be solved are the *mean* (μ) and seven effect coefficients $\{\beta_i, \beta_{ij}, \cdots\}$. While these eight unknowns do require eight observations of R (trials) for the solution of equation (1), a full 2^p set of trials is usually not needed when the number of factors, p, becomes larger. Many terms are then assumed to be of no consequence and not solved for. Such trade-offs within common DEX designs have been tabulated (see Table 12.15, [2], p. 410). Weak terms of little influence in equation (1) will have effect coefficients relatively close to zero. In practice, some effects are insignificant, but not all. Interactions are less common and higher-order interactions, rarest. This is especially likely for a large number of factors. Thus, X_{ABC} is usually less influential than X_{AB} or X_{AC}. DEX practice often assumes few interactions but sets safeguards should the assumption fail.

Effects are commonly expressed as column entries in a table (see lower left of Figure 1), a more convenient format for screening comparisons than is equation (1). (In screening, a predicted response R is not the first concern.) The row corresponding to an effect identifies its source(s). Effects are evaluated against noise, the latter often expressed as *standard uncertainty*, S. The handling of trials, such as whether to replicate, is dictated by the degree of confidence required in the tuning process. The two examples in the text use replicated trials for estimates of S. Coefficient noise in the model, (1), has a normal distribution centered about zero with an *estimated* standard deviation of S, the standard uncertainty. Consequently, 68.3% of all coefficient noise will fall within $\pm 1S$, 95.4% within $\pm 2S$ and 99.7% within $\pm 3S$. A significant effect will probably exceed 3 or $4S$. Comparisons among significant effects establish which factors most influence response R. (Figure 1 shows a $3S$ threshold.) Response R varies about an overall mean μ shown in (1). μ is an estimate of R with all treatment settings at halfway, *i.e.*, $X_A = X_B = ... = 0$. (Remember that $X = -1$ *or* $+1$ for the settings.)

3. SPT on Shared and Distributed-Memory Systems

Two small but quite representative programs illustrate the essential SPT with a minimum of detail and complication. These 300-400 line examples run on shared-memory (Example 1-SM) and on distributed-memory (Example 2-DM). In practice, SPT has been applied to codes 20 to 30 times larger.

3.1 Example 1-SM: Tuning a Parallel Quicksort

Shared-memory architectures are generally more common and better balanced than distributed-memory systems. In this first example, SPT is applied to a parallel sort algorithm on a conventional, well-balanced shared-memory multiprocessor (16 processor Sequent Balance system). The investigation begins with application of the UNIX profiling tool, *gprof* [3]. Once a determination of base performance has been made, SPT tunes the application in one experimental step of 16 trials (replicated). Performance improves by 23%. The investigation reveals differences between *gprof* and SPT techniques and contrasts results from each. A second SPT iteration exemplifies possible stopping circumstances. The program is a parallel implementation of C.A.R. Hoare's *quicksort* [4]. Parallelism is obtained simply by allocating newly-created sublists to available processors. Allocation of these sublists is controlled by a shared stack. Whether this is the best algorithmic recoding is not a direct SPT issue.

3.1.1 An Approach via the Tool *gprof.* Investigation begins by seeking information with the tool *gprof.* Most UNIX-based systems provide *gprof* to generate profiles of programs. Profiling tools help in debugging and in improving efficiency. Some functions consume significant execution time. Others are called frequently. Once important functions are identified, their code can be improved. This paradigm has been successful for tuning sequential programs running on uniprocessors. However, emphasis changes on a parallel architecture. Process interaction and processor idle time play a vital role in the performance of parallel programs. These are absent or insignificant in most sequential programs. A simple extension of a sequential profiler on a multiprocessor can measure the total time a segment of code spends on each processor.

This is one possible measurement metric. However, a code segment's total processor time is not related in a simple way to parallel runtime (Anderson and Lazowska [5] discuss this). If the results from a profile are interpreted incorrectly or if a profile is not available, a programmer can waste time and effort improving code that has little impact on overall performance.

code segment	% of execution time	# of calls
s_lock()	64.3	359031
pop()	0.1	13183
push()	0.2	13183
swap()	5.1	811141
bubble_sort()	3.8	6592
code1	na (not avail.)	na
select()	0.4	6591
main()	7.9	1
partition_list()	11.3	6736
s_unlock()	1.3	359031

Table 1: Abbreviated *quicksort* Results from *gprof*

Table 1 (above) gives abbreviated results from a parallel *gprof* file for the *quicksort* routine. Two important metrics from the profile are (1) the percentage of execution time consumed by a routine and (2) the number of instances the routine was called. For metric (1) it is clear that a majority of time for this parallel *quicksort* is spent in the s_lock() routine. In addition, execution time mildly suggests four other routines: bubble_sort(), swap(), main(), and partition_list(). Regarding metric (2), only swap(), s_lock(), and s_unlock() stand out with high invocation counts. The programmer must decide from this information where to make changes. The task would be more obvious on a uniprocessor; generally, any time saved would be reflected in a shorter response.

Unfortunately, *gprof* data are murky in a parallel domain; interactions in the program structure and concealed wait states have to be accounted for (*e.g.*, [6]). Perhaps the programmer will improve s_lock(), since it consumes the majority of overall runtime. Routines bubble_sort(), swap(), partition_list(), and main() may be overlooked; improving these routines seems to offer little benefit. Analyzing the data from a call graph perspective focuses attention upon swap(), s_lock(), and s_unlock(). It is not easy to relate invocation counts to runtime effect in a parallel domain. It may be that a simple profile of a parallel program offers no clear plan of attack; alternately, it might encourage a wrong interpretation of results.

3.1.2 New Approach. SPT testing of the parallel *quicksort* begins within the familiar framework of a 2^{6-2} fractional factorial experiment design [2]. There are $2^4 = 16$ trials. Results from the analysis appear in Table 2 (see page 11). The first column is the observation (trial) identification (trials run in random order). The next six columns designate treatment settings for six factors chosen in the experiment. The six factors (f1 to f6) are (from left to right): s_lock(), push(), pop(), swap(), bubble_sort(), and code1. Although the first five factors are function calls, code1 is a segment of code (a loop) within the main *quicksort* function. SPT can resolve coarser or finer, and is not restricted to function calls. A plus sign in a given row denotes that treatment is set (*i.e.*, delay present). Minus denotes treatment absent (*i.e.*, no time delay). Thus row 5 (− − + − + +) shows f3, f5 and f6 (pop(), bubble_sort(), code1) with delays set. The response column, *Rspn*, is the average of three separate trials for each row's treatment combination. Error estimates, *S*, also arise from these trial replications. An "Effects" column gives the mean, μ, and effects $\{\beta\}$. The "Sources" column lists the *most likely factors* for an effect. Some confounding (mixing together of factor effects) has been deliberately introduced to shorten testing (see [2]).

It is clear from line 9 of Table 2 that swap() is least tolerant of source code perturbation. The significance of $\beta_{swap()}$ is hard to doubt, since $\frac{\beta_{swap()}}{S} = 501$. A survey of remaining effects indicates no other outstanding combinations. In contrast to earlier s_lock() indications with *gprof*'s metrics, the effect $\beta_{s\_lock()}$ in line 2 of the analysis is subtle and weak. Associated with non-productive wait states, s_lock() is not the source of the major bottleneck. SPT points first and foremost to swap(). An examination of the swap() routine reveals that it is very short. Coding swap() in-line frees it from procedure-call overhead, perhaps its major execution cost. The result is a one-step, 23% boost in *quicksort*'s performance.

trial	f1	f2	f3	f4	f5	f6	Rspn	Effect*	Sources
1	-	-	-	-	-	-	19.11	48.03	Mean, μ
2	+	-	-	-	+	-	19.99	1.98	s_lock()
3	-	+	-	-	+	+	21.73	1.12	push()
4	+	+	-	-	-	+	26.51	0.66	s_lock()&push()
5	-	-	+	-	+	+	21.91	1.30	pop()
6	+	-	+	-	-	+	26.42	0.66	s_lock()&pop()
7	-	+	+	-	-	-	21.51	-1.24	push()&pop()
8	+	+	+	-	+	-	26.60	-0.10	s_lock()&push() &pop()
9	-	-	-	+	-	+	72.75	50.10	swap()
10	+	-	-	+	+	+	72.97	-1.84	s_lock()&swap()
11	-	+	-	+	+	-	73.32	-1.12	push()&swap()
12	+	+	-	+	-	-	72.66	-0.46	s_lock()&push() &swap()
13	-	-	+	+	+	-	73.49	-0.98	pop()&swap(), OR bubble_sort()
14	+	-	+	+	-	-	73.13	-0.32	s_lock()&pop() &swap()
15	-	+	+	+	-	+	72.52	1.10	push()&pop()&swap(), OR code1
16	+	+	+	+	+	+	73.81	0.74	s_lock()&push()&pop() &swap()

* Standard Error for effects: S = ± 0.10
Standard Error for the mean: ± 0.05

Table 2: Calculated Effects for 2^{6-2} Factorial Design, Parallel *quicksort* Example

	Mean	
	27.52	
Source(s)	Effect	
s_lock()	6.50	<--1
push()	4.26	<--*
s_lock()&push()	1.36	
pop()	4.46	<--*
s_lock()&pop()	1.36	
push()&pop()	.48	
s_lock()&push()&pop()	-.46	
bubblesort()	-.08	<--2
s_lock()&bubblesort()	-.36	
push()&bubblesort()	-.72	
s_lock()&push()& bubblesort()	.80	
pop()&bubblesort()	-.58	
s_lock()&pop()& bubblesort()	.54	
push()&pop()& bubblesort()	1.28	
code1 OR s_lock()& push()& pop()&bubblesort()	4.46	<--*

```
Standard Error of an effect:  S = ± 0.08
Standard Error of the mean:  ± 0.04
```

Table 2-B: Effects for 2^{5-1} Factorial Design, Improved Version, Parallel *quicksort*

A second iteration of SPT demonstrates a better balance in the modified *quicksort*. The new version with in-line swap() is run in a 2^{5-1} design. Swap() is not tested. In examining Table 2-B, above, the reader should know that the data, synthetic delay function and experiment design differ from those used for Table 2: This precludes any direct comparison across the two tables without rescaling (discussion follows). The largest effect in Table 2-B is indicated by <--1. Unfortunately, it belongs to s_lock(), a system function that the programmer cannot change easily. The recourse is an algorithmic redesign that uses less of s_lock(). Although this would lie beyond what SPT can recommend, SPT can certainly assist in the selection. The factor bubblesort(), shown as <--2, again has little effect overall and can safely be ignored. The three remaining, user-accessible factors--push(), pop(), and code1--have an almost perfect balance among their effects. If the sort's speed is adequate at this point, the programmer may want to stop. No single recoding improvement emerges among the factors. Main effects for the improved *quicksort* (*qs*+) and the original (*oqs*) have been normalized in Table 2-C (below) as percentages of unperturbed runs *for each sort version.*

β,oqs	β,qs+	Sources
10.36	14.86	s_lock()
5.86	9.74	push()
6.80	10.20	pop()
262.17	---	swap()
-5.13	-0.18	bubblesort()
5.76	10.20	code1

Table 2-C: Normalized Main Effects Show Sensitivity Changes

3.1.3 Shared-Memory Results. SPT unequivocally distinguishes a major performance bottleneck in the parallel *quicksort*. In contrast, a micro-level profile tool such as *gprof* can confuse true performance contributions with unproductive busy waiting; this distorts *gprof*'s sense of what code is important. The newer tool, Quartz [5], tries to improve upon *gprof* by dividing all profile times by the average level of parallelism for each profile category. This diminishes emphasis upon categories that are highly parallel and have little room for improvement via concurrency. A Quartz-like evaluation might reduce s_lock()'s accounted time in Table 1. The most optimistic circumstance of full 16-level parallelism would give $\frac{64.3}{16} \approx 4$. But s_lock() would still have a standing roughly the same or higher than swap(), for swap() also has attendant concurrency.

In comparison to the largely constructive, micro-model approach of metric methods (*gprof*, Quartz), SPT is more straightforward. It simply avoids structural metrics and their interpretations. Comparing columns of results of *gprof*-like tools with SPT's results yields important distinctions. Numbers from *gprof* are raw measurements of interior (factor) detail. The user must constitute models that predict a program's responses. SPT is exactly the reverse. Program responses are first measured and then correlated against precisely controlled changes in factors. Consequently, the SPT list of effects for factors is anything but raw data. Effects in the SPT model (equation (1)) beckon to exactly those code locations with greatest impact upon performance. The *quicksort* example shows SPT to be very accurate in identifying real bottlenecks in a program on a well-balanced parallel system.

3.2 Example 2-DM: Ring-Connected Nodes

Distributed-memory systems present definite challenges to designers of Quartz-like tools. Compared to shared-memory, the measure *level−of−parallelism* is significantly more difficult to capture as a distributed-memory statistic. In contrast, SPT works as usual. SPT uses macro-level analysis and does not rely upon precise internal system details.

The distributed-memory "Ring" example exhibits a broader than usual set of SPT gains. It is an exaggerated object lesson for the host architecture. A more ordinary example (*e.g.*, a benchmark "Mesh" for fluid-like computations) requires the same approach as this example, but it will not yield improvements quite so drastic. The program slice in Figure 2 (below) is for an iPSC-1 distributed-memory hypercube system. Temporarily ignore the six invocations of *delay*. Each of the system's 16 processor nodes contains a copy of the code sketched in Figure 2. All the hypercube nodes have been programmed as a single ring; each Node Program B sends its main messages to one unique node and receives from another. At the start, each node sends a main message. Thereafter, the outermost "B:loop" awaits main messages via a RECVW. It passes each message to an <if-statement>. Each branch of the <if-statement> also awaits a secondary message for flow control. The TRUE branch is much more likely.

```
   ...<start>...
B: loop ...
     delay(F1); RECVW(); delay(F2); /* get main msg */
     ...
     if(...)
          ....
          loop
             ..  {code A} ..
          endloop
          ...
          delay(F3); RECVW(); delay(F4)
     else ...
          delay(F5); RECVW(); delay(F6)
          ...
     SEND();                           /* send main msg */
endloop: B
```

Figure 2: Node Program B--Slice with Perturbations F1-F6

Following SPT's steps 1-5, the investigation again proceeds in stages of DEX refinement and elimination. Initial screening of Node Program B is via a first set of delays at code locations labeled *a* through *h*. These are not shown in Figure 2, but appear in Table 3 (below) as source labels for effects. Delay levels of treatment are 0 (rescaled as −1) and 10 (rescaled as +1). Factor c, which corresponds to code segment {code A}, is known to be computationally important. Other segments are simply chosen uniformly throughout B. The experiment uses a 2^{8-4} resolution IV design (see [2]). This design confounds main effects with three-way interactions, the latter assumed to be negligible. Second-order effects are confounded with each other (see the Source column in Table 3). There are $2^{8-4} = 16$ trials. Trials are run with the test code configured for normal service of light communication and heavy computation. {code A} is by far the most sensitive (arrows in Table 3). An independent check (by varying intrinsic program parameters) reveals that a 10% increase in {code A}'s execution time causes a 7% increase in overall program response time. This is true even though {code A} is very short (one multiply-and-assign within a for-loop). Consequently, a warning comment should be placed in the program stating that changes to {code A} should be done with care. This experience is similar to the earlier *quicksort* example. However, *negative* effects in Table 3 hint there is more to be learned.

		Mean	
		26.75	
	Source(s)	Effect	
	a	-2.25	..?
	b	-2.50	..?
	ab+cg+dh+ef	2.75	
-->	c	18.50	<-- {code A}
	ac+bg+df+eh	2.25	
	ag+bc+de+fh	2.50	
	g	-2.75	..?
	d	3.00	
	ad+bh+cf+eg	-2.75	
	ah+bd+ce+fg	-2.50	
	h	2.75	
	af+be+cd+gh	-2.50	
	f	3.25	
	e	3.00	
	ae+bf+ch+dg	-2.25	

```
Standard Error of an effect:  S = ± 0.50
Standard Error of the mean:  ± 0.25
```

Table 3: Node Program with Delays a-f

3.2.1 Optimization Rather than Screening. The most significant effect in Table 3 is β_c, which is 18.5/0.5 = 37 standard uncertainties (deviations) removed from 0. It is unlikely that c's *true* effect is zero and that c's *observed* effect arises from sampling noise. In contrast, other delays at a, b and g show weaker but *negatively* signed effects. These delays appear to promote shortened overall runtimes, an unusual circumstance (see "..?" in the table). At four or five standard uncertainties in magnitude, these unusual delay effects are worth investigating. Perhaps delays can make the "Ring" run faster.

Delays a, b and c (not shown in Figure 2) are near synchronous receives (RECVWs) in Node Program B. To investigate further, a second experiment is devised with a fresh copy of the program that has delays before and after all RECVWs. Figure 2 shows this version with its delays F1 through F6. The program is now configured with parameters for heavier communication and lighter computational load, since prior experience has shown communication congestion can be a problem. With delay(0) everywhere (no perturbations), the program takes 66 seconds. A two-level (treatments of 0 and 10), 2^{6-2} resolution IV factorial experiment of 16 trials uncovers an improvement of 37% with F1=F2=F2=10. However, different delay treatment levels uncover strong signs of nonlinearity in the program response. This triggers two methodology changes: (i) more than one level of delay seems appropriate in the investigation, and (ii) the linear response model is abandoned for *this example.*

A sequel experiment is performed in which F1-F6 have five possible levels: X=0, 7, 14, 21 and 28. Although this might entail 5^6 possible combinations, the experimental design (called an $OA_{25}[5^6]$ layout) uses only 25 trials in its search [7]. Sample results follow in Table 4.

trial#	F1	F2	F3	F4	F5	F6	Result	notes
1	0	0	0	0	0	0	66	Normal Case.
	...	...						
7	7	7	14	21	28	0	47	
	...	...						
16	21	0	21	7	28	14	33	Twice as Fast.
	...	...						
25	28	28	21	14	7	0	50	

Table 4: Complex Search of Delay Settings

Linear effects are not calculated. Instead, the $OA_{25}[5^6]$ layout is used simply to search the parameter space of delay settings in a geometrically balanced way. The chosen trial has the best response (see [8], pp. 386-389, on "Informal Methods"). This straightforward approach can be quite effective, as one can see with the response of 33 seconds (trial number 16). Certain *permanently installed* delays enhance Program B's speed. The explanation lies with the iPSC-1 system. Heavily used message-passing provokes communication failures, which then require retransmissions. Delays help synchronize nodes and cut the wasteful communication congestion. However, given the nonlinearities involved, discovering a very good setting (such as 21-0-21-7-28-14) is non-trivial. Conventional linear models, even contour plots, seem to be of little value [9]. If only naive methods are employed, the challenge will probably elude all but the most determined practitioners. DEX's rational and efficient search methods help greatly. In practice, Node Program B of Figure 2 would be adaptive, making delays when heavily communication bound and omitting them otherwise.

3.2.2 Distributed-Memory Results. The experiment for distributed-memory shows that under ordinary circumstances, SPT indicates which code sections are most sensitive to delays, and equally important, which sections are insensitive and easily modified. {code A} is to Node Program B as swap() is to *quicksort*. However, when an iPSC-1 program is communication bound, delays can sometimes enhance performance. In this third case, the synthetic perturbations transcend their role in the investigation to become part of the solution. Such cases are likely to be increasingly rare, because over time architects will strive to remove obvious system bottlenecks.

4. Different Delays, Distinct Systems

This last section explores some questions that arise naturally about the choice of delay values. Example 2-DM has demonstrated clearly that the level of delay matters in some special circumstances. However, even in everyday cases this may be true. Because actual treatment levels are rescaled to – and +, there may be a tendency to overlook values of delay treatment. A third example program will demonstrate that even a generally well-behaved code can display quite distinct sensitivities on different architectures. For the shared-memory case, delay values above a critical level yield essentially the same results. The distributed-memory variant is not so well-behaved.

The delays added to a program must be large enough to surmount noise inherent in the experiments. In the terminology discussed earlier, S is the standard uncertainty, an estimator for the standard deviation of an effect. Let $3S$ denote an *expanded uncertainty*, a noise band; expanded uncertainty is often chosen at $2S$ (95+% significance), but here $3S$ (99+% significance) works well and identifies clear-cut bottlenecks. Delay treatments should be chosen such that truly important effects exceed $3S$. Otherwise, an effect lost in noise cannot be separated from it. As will be seen shortly, both delay setting and choice of expanded uncertainty depend upon objectives within the investigation; studies of subtle phenomena will require lower settings for both.

The vehicle of exposition is Mesh, a computer benchmark running on both the shared-memory Sequent (example 3-SM) and the iPSC-1 hypercube (example 3-DM). In Mesh's computations, each logical node at a mesh intersection receives and uses state information from four logically immediate neighbors to update its state. Fluid computations may have this character. The concrete topology for the actual tests is a 4-by-4 grid with edges wrapped around. This gives each logical node a real processor on both machines, each of which has 16 processors. Neighbors to any node are one step apart on the iPSC-1 hypercube, so all communication avoids intermediate nodes. The shared-memory version does not worry about allocation, since all addresses are equally reachable.

4.1 When Delay Choice Matters Less: Example 3-SM

A shared-memory experiment with Mesh demonstrates what effects from different sizes of delay look like under benign circumstances. On a system where effects are truly linear, as they are here, a bigger delay (treatment) of a significant factor will cause a larger change (effect) in the observed response. In Figure 3-a (follows), Mesh is running on the shared-memory Sequent as example 3-SM. Notice how progressively larger delays generate linearly proportional changes in the two observed main effects. Under such circumstances, the major concern is that delay size be chosen adequately large *vis−a−vis* expanded uncertainty. Beyond this 3S threshold, the size is unimportant, because the flat slope means that effect *per unit delay* is constant and predictable. Consequently, the researcher can be relatively casual in selecting a delay setting. The setting certainly matters much less than in the distributed-memory case, which follows.

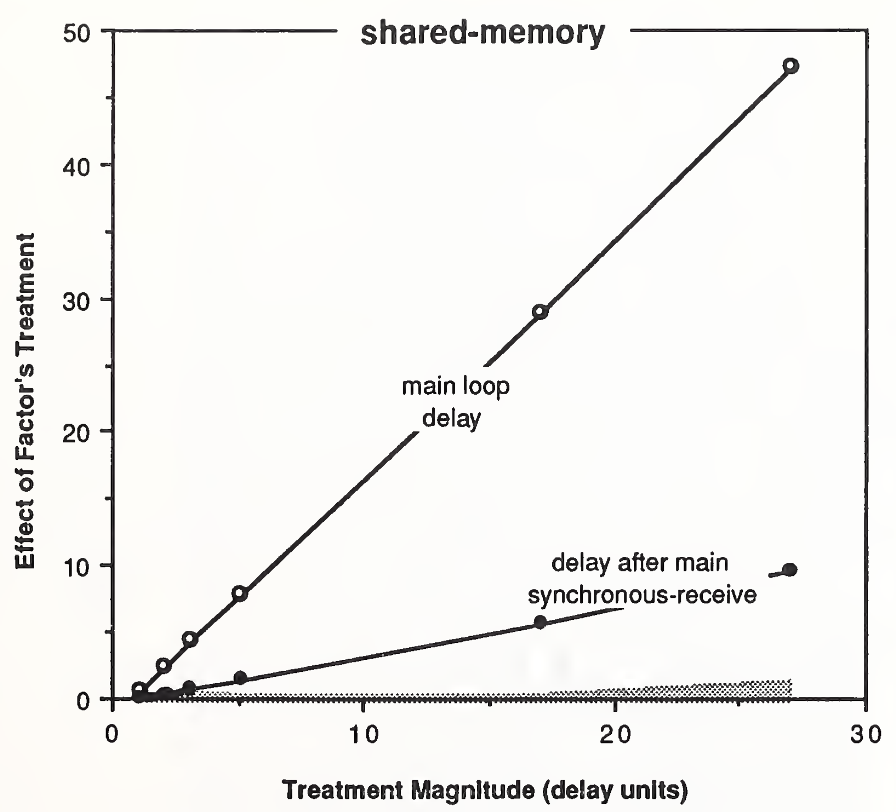


Fig. 3-a

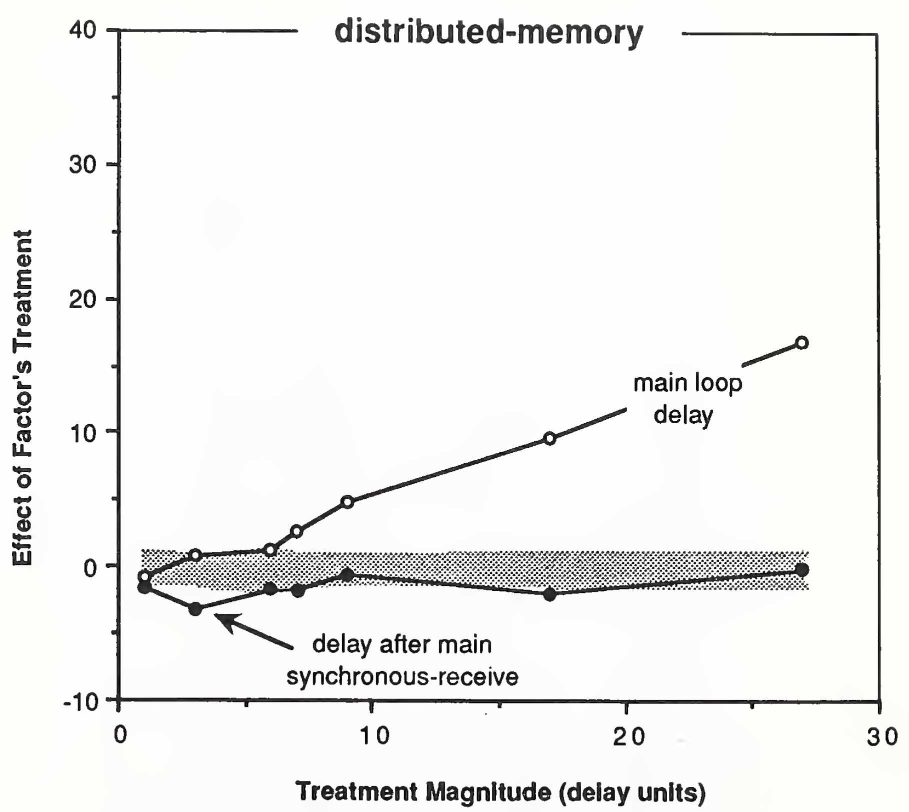


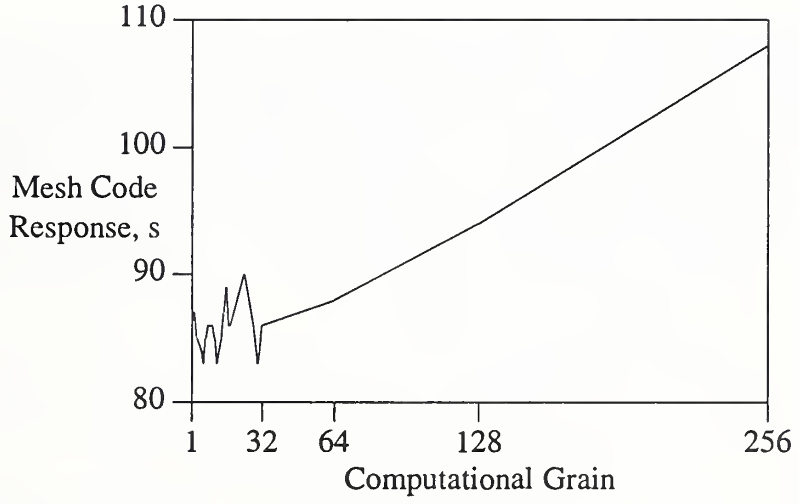
Fig. 3-b

Figure 3: Two Main Effects of "Mesh"

noise=expanded
uncertainty= ▨

4.2 Higher Unpredictability: Example 3-DM

Given earlier experience with Ring example 2-DM, the distributed-memory version of Mesh is also a likely source of non-linear behavior. To explore this possibility quickly, a natural program parameter was systematically changed and the responses compared. (Not all programs are so obliging.) Mesh has a parameter for *grain*, the amount of computing done for each state, which in turn is proportional to the computing done upon each message datum. The figure below shows typical Mesh response times as grain is increased. Observe that up to grain=32 ($x-axis$), Mesh is erratic in response. This is a nonlinear regime in which delays and their attendant responses might not obey the assumed linear response model. Investigation thus breaks into at least two main parts, the linear region of larger computational grains, and a nonlinear region where communication heavily influences program behavior.



With suspicions raised about the Mesh variant 3-DM, a conscientious effort must now be made to check observed effects against a spectrum of delay values. Figure 3-b (page previous) shows these results. First, observe that uncertainty (the noise band) is higher on this implementation. Also, while the strongest effect (main loop delay) is similar for both architectures, it is weaker for the hypercube version (3-DM), with a less certain behavior for smaller delays.

The second main effect shows that small delay treatments of its factor actually accelerate execution. This is analogous to behaviors seen with the earlier Ring (2-DM), but on a reduced scale. Once delays reach five nominal units, the second factor's effect becomes insignificant. Altogether, the results show an expected program computational slowdown from larger delays, and an unconventional system speedup from smaller perturbations. The lesson for 3-DM and similar cases is that the experimenter must have some knowledge about sizes of perturbation, and from this adjust the delay treatment to suit factors of interest.

5. Conclusions

Statistical screening is different from building a detailed performance model, *e.g.*, the FORTRAN virtual system in [10]. SPT applies DEX screening to avoid the inefficiency of one-factor-at-a-time examinations as it searches for important effects. This use of screening is not unique. In computer system tuning, several gross hardware or operating system factors may be systematically treated and screened by substitution at the module level [8]. Examples are memory at 1 Mbyte/processor *versus* 4 Mbytes/processor, or file transfer via FTAM or FTP. But SPT examines applications in code-level detail. Rather than several factors, there may be several *hundred*. Consequently, component substitution for each application factor becomes thoroughly impractical. System tuners may have little practical choice but to use *ad hoc* substitution treatments. They often work within tightly constrained circumstances. Fortunately, SPT has more latitude. Real-time applications aside, source code is very tolerant of inserted synthetic perturbations. By deliberately avoiding component substitution, SPT treatments are *uniform*, which encourages automatic testing. Computer controlled scripts can generate and schedule experiment trials.

A large distributed-memory system may have difficulty capturing global information for metrics at fine enough resolution. SPT avoids this problem by not depending upon detailed performance metrics. Similarly, interpretation of detailed metrics can be uncertain. Change in some state may induce a corresponding change in response time that exceeds explanation via level-of-parallelism, the highest multiplier one might first expect [6]. However, interprocess communication can render process states highly interdependent. The combinatoric nature of the interdependencies strains analysis. SPT successfully skirts this problem by approaching the application and system as a complex, poorly understood entity. Structural complexities and interpretations within program or system then matter far less.

Since it needs little internal detail, SPT makes no demands for special measurement resources such as fast clocks, counting registers or on-line data collection. Coarse global timing can provide good results: a one-second clock tick is serviceable. The distributed-memory example, 2-DM, uses a one-second tick for its results. Alternately, better instrumentation combined with SPT opens many exciting new opportunities. Analogous to its modest need of instrumentation, SPT in its basic form has a low visual demand. Simple character-display screens suffice. Tables 2 through 4 are typical SPT displays.

The table below depicts some differences between conventional micro-level and the SPT macro-level tuning paradigms:

	Micromodel	Macromodel/SPT
Factors	*measured (metrics)*	*fixed (in patterns)*
Overall Response	*derived*	*measured*
Model	*fixed (construct)*	*derived (effects)*
Basis	*theoretical*	*empirical*

5.1 Further Directions

While the two examples in the text illustrate essential aspects of SPT, the technique opens a rich field of possibilities and challenges. Consider the question of program size. The text examples are small, around 400 lines. Applications are usually larger. The largest SPT test has involved 12K lines on a shared-memory architecture. At this magnitude, source code presents very real and practical questions on the choice and handling of delay sizes, and especially, the identification and handling of a large number of factors. For example, the statistical approach for large programs may shift away from factor interactions as they become less likely. Instead, emphasis is given to screening many factors (64-128) in each iteration step. Initial large program experiences indicate that much effort is spent within the screening analysis loop (steps 1-5). Code improvement (step 7) takes a smaller fraction of the effort.

Eventually, SPT will have a programming environment to support its features. An intermediate step will be a library of tool set routines. These will generate experiment layouts, support simple analysis and generally relieve some of the tedium of handcrafted setups.

5.2 Summary

SPT combines synthetic delays with DEX to yield an attractive new technique for MIMD program improvement. DEX lends to SPT some formidable powers of search and analysis, while in turn SPT's synthetic treatments render DEX setups and trial variations much faster and more convenient. Both shared-memory and distributed-memory MIMD architectures are suitable hosts.

Acknowledgments

Thanks go to C.S. Lyon, A. Mink, A. Shives and T.E. Wheatley for suggestions on earlier versions of the text, and to colleagues at the Supercomputing Research Center, Bowie, MD., for their willingness to entertain and critique our approach.

6. References

[1] G. Lyon. Application on synthetic perturbation tuning, Patent and Trademark Office, Washington, DC.

[2] G.E.P. Box, W.G. Hunter, and J.S. Hunter. STATISTICS FOR EXPERIMENTERS. John Wiley & Sons (New York, 1978).

[3] S.L. Graham, P.B. Kessler, and M.K. McKusick. "Gprof: A Call Graph Execution Profiler." Proc. ACM SIGPLAN Symposium on Compiler Construction, June, 1982.

[4] C.A.R. Hoare. "Quicksort." Computer Journal 5, 1(January 1962), 10-15.

[5] T.E. Anderson and E.D. Lazowska. "Quartz: A Tool for Tuning Parallel Program Performance." Proc., SIGMETRICS 1990 Conf., May 1990, 115-125.

[6] R. Snelick. "Performance Evaluation of Hypercube Applications: Using a Global Clock and Time Dilation." NISTIR 4630, July, 1991, 26 pp.

[7] R.N. Kacker, E.S. Lagergren, and J.J. Filliben. "Taguchi's Orthogonal Arrays are Classical Designs of Experiments." J. Res. Natl. Inst. Stand. Technol. 96, 5(Sept.-Oct. 1991), 577-591.

[8] R. Jain. The Art of Computer Systems Performance Analysis. J. Wiley & Sons (New York, 1991), 720 pp.

[9] J. Filliben. Private conversations with and notes to authors. November, 1991, NIST.

[10] R.H. Saavedra-Barrera, A.J. Smith, and E. Miya. "Machine Characterization Based on an Abstract High-Level Language Machine." IEEE Trans. on Computers 38, 12(Dec. 1989), 1659-1679.