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

NISTIR 4859

Computer Systems Laboratory

Time-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

CMRF

COMPUTER MEASUREMENT
RESEARCH FACILITY
FOR HIGH PERFORMANCE
PARALLEL COMPUTATION

June 1992

Partially sponsored by the
Defense Advanced Research Projects Agency

NISTIR 4859

Time-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
Arlington, VA

June 1992

U.S. DEPARTMENT OF COMMERCE
Barbara Hackman Franklin, Secretary

Technology Administration
Robert M. White, Under Secretary for Technology

National Institute of Standards and Technology
John W. Lyons, Director

TABLE OF CONTENTS

Page

Time-Perturbation Tuning of MIMD Programs

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

Time-perturbation tuning--TPT--is a novel technique for assaying and improving the performance of programs on MIMD systems. Small, synthetic program delays are combined with statistically designed experiments (DEX). Claims for TPT are two: (i) Conceptually, it brings the powerful, mathematical perspective of experiment design to interdependent, sometimes refractory aspects of MIMD program tuning; (ii) Practically, it provides a needed speedup mechanism, synthetic time 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. TPT 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.

 Partially sponsored by the Defense Advanced Research Projects Agency, Task No. 7066, Amendment 03.

1. A Problem with MIMD

MIMD programs are difficult to code well and to improve--the asynchronous, interdependent nature of 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. True improvements are not always intuitive in any serial sense of the code. For example, communication-induced idle time exerts a big influence (illustrations follow). It quickly becomes clear upon reflection that scalable parallel systems can present an avalanche of internal detail. Most of this information should be kept hidden from users. Yet, MIMD programmers need to identify program bottlenecks. This raises the question of how users are to improve their programs. Conversely, programmers should know which sections of code can be expanded with only minor performance penalties. Although isolated changes in MIMD code certainly can be monitored for overall performance results, a typical parallel program will have a large number of potential improvement points. The effort in any naive attempt at a comprehensive evaluation is simply too great.

2. Time-Perturbation Tuning

There is an alternative, more in the spirit of transparent system service, that suggests perturbing programs with systematic pattern sets of artificial delays. This is time-perturbation tuning, TPT. The sets of delays are easily generated and inserted into code, since they are synthetic quantities divorced from normal computational states. Response times are measured for runs performed with each pattern of delays. From this is built a macro-level model that predicts program sensitivity to delay settings. Internal details of a program are largely ignored, a philosophy that contrasts sharply with conventional tuning approaches. TPT rests heavily upon the design of experiments (DEX), a modern branch of statistical theory well-suited for studying complex systems [BHH78]. DEX provides *quantitative* estimates of the effects of TPT's synthetic bottlenecks. The objective is to establish quickly and accurately which locations matter most for improvement. Although a location can be on a control path or a data path, control paths are emphasized in this discussion.

2.1 Description of Technique

Time-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 perturbation that are inserted into program *P*. TPT follows the following steps:

1. Select code locations in program *P* to be tested, via synthetic delays, for their effect upon performance.
2. Insert standard perturbations in statistical test patterns. This will generate numerous treated versions of the program. Order these randomly.
3. Run each treated version of the program and record its response. Replications should use a fresh randomization order.
4. Analyze the measured responses. *ANOVA*, the analysis of variance, is typically employed. In some circumstances, the system response surface is not well behaved. Here, more informal methods may prove appropriate (an example follows in the text).
5. Assess the analysis and select those effects that are significant. These effects correspond to factors and their interactions. Insignificant factors are dropped and new factors added.
6. Repeat steps 1-5 as needed. Analysis proceeds in cycles of refinement.

Selecting initial factors in a program requires judgment on the nature of the software pieces. This first estimate can be refined as sensitivities become known. Since TPT assays quantitatively, influential factors can be retained and investigated further alongside promising new ones.

2.1.1 Treatment Opportunities. There are numerous ways to treat the code. 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. It should be noted in passing that the chosen *treatment* does not have to be a time delay, although the latter is often first to come to mind. 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 delay possibility for source code is to insert them along *data paths*. An illustration for such a delay treatment may be useful. Let integer variable *A* be a location of interest. References to *A* are replaced 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 treatment lie system-level approaches. Here, the available resources will determine what can be done. Processors that can trap on 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. This would be the case if treatment meant code had to be tediously rewritten to be slower or faster while giving the same results. DEX treatments of programs, typically system pieces, have had this nature in the past. Fortunately, such rewriting is unnecessary, for synthetic delays are deliberately chosen to be outside the logical (or computational) state of the program under test. For example, the delay function does not use global names or values from the original code. Logical, or computational, soundness of a program remains as it was originally (exclude real-time codes). For this reason, perturbation points can be swiftly inserted, tested and removed. The perturbation used at all points is standard--it is not tailored *ad hoc* for each context. This fact is pivotal to practical applications of TPT, including the automatic setup of trials.

2.1.2 In Practice. Each inserted delay simulates added instructions. Code for the standardized delay need be neither large nor complicated--six to fifteen machine instructions often suffice. In the distributed-memory experiment that follows (Example 2-DM), delays are generated by a recursive function, *delay*(*X*), embedded in a compilation directive that is conditional upon *integer* parameter X. X=0 generates no 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.

Patterns of delays are determined once p program test factors (locations) have been chosen. As will be seen shortly, the TPT approach starts by treating each factor at two delay settings. Given the p factors, an exhaustive experiment will thus involve 2^p patterns. This is a *full factorial* design. *Partial factorial* experiments, designated 2^{p-k}, generate far fewer patterns, but have less experimental resolution. Partial factorials are generally quite adequate. Treated versions of the test program P are run with their pattern of delays. These runs are *trials*. An overall *response* is measured for each trial. 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 type of response corresponds to a distinct experiment. Thus each experiment takes multiple trials, but the same trials can serve multiple experiments. Once a set of trials is complete, the statistical method *analysis of variance*, *ANOVA*, is commonly employed.

2.1.3 The Screening Model. TPT incorporates a screening method. More so than other statistical approaches--regression, optimization, comparison--a screening focuses upon identifying which among tested categories matter most. TPT's analysis rests upon a simple 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 interaction $X_{AB} = (-) \times (+) = -$. These binary settings are generally adequate, although exceptions are explored later. Still, circumstances suggest first investing only a minimum in each model, since program tuning will change the very code under study. Model utility 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:

$$R = \mu + \tfrac{1}{2} [\beta_A X_A + \beta_B X_B + \beta_{AB} X_{AB} + \beta_C X_C + \beta_{AC} X_{AC} + \beta_{BC} X_{BC} + \beta_{ABC} X_{ABC}] \quad (i)$$

Multiplier ½ arises because the domains of treatment variables $\{X_j\}$ span $[-1, +1]$, a distance of two. The unknowns to be solved are the mean (μ) and coefficients $\{\beta_i, \beta_{ij}, \cdots\}$. As mentioned, p factors generate 2^p unknowns, a formidable number for larger p. It is important to note that DEX emphasizes ascertaining which among a model's terms are significant, and which can be safely ignored, without exhaustively exploring all 2^p combinatoric trials. Partial factorial experiments 2^{p-k} deliberately *confound* effects that are thought unimportant. Such tradeoffs among DEX designs are well known (see Table 12.15, [BHH78], p. 410). Weak terms of little influence will have

coefficients closer to zero than do the significant factors. In practice, many terms are insignificant, but not all. Interactions are less common, and higher-order interactions, rarest. This is especially true if there are many factors. Thus X_{ABC} is usually less influential than X_{AB} or X_{AC}. DEX practice will often assume few interactions, but set safeguards that signal should the assumption fail.

Coefficients of the model (β_A, β_{AC}, *etc.*) are its *effects*. Effects are commonly expressed as column entries in a table, a more convenient format for screening comparisons than is equation (i). (In screening, a predicted response R is not the first concern.) The row corresponding to an effect identifies its source(s). Effects are evaluated relative to model noise, the latter often expressed as *standard error*, S. Handling of the trials, such as whether to perform replicates (duplicate runs), is dictated by the degree of confidence required in the tuning process. Examples herein use replicated trials for estimates of S. Noise in the model, (i), has a normal distribution centered about zero with an estimated standard deviation of S. Consequently, 68.3% of all noise effects will fall within $\pm 1S$, 95.4% within $\pm 2S$ and 99.7% within $\pm 3S$. A significant effect will probably exceed 3 or 4 S. Comparisons among significant effects establish which factors most influence response R. Response R varies about an overall mean μ shown in (i). 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. TPT on Shared and Distributed-Memory Systems

Discussion proceeds via two small but quite representative programs, the first running on a shared-memory architecture, and the second on distributed-memory. These 300-400 line examples illustrate the essential TPT with a minimum of detail and complication. In practice, TPT has been applied to programs of up to 12,000 lines of code.

3.1 Example 1-SM: Tuning a Parallel Quicksort

If TPT is to be useful, it must be effective in everyday practice. Shared-memory architectures are generally more common and much better balanced than message-passing systems. In this first example, TPT is applied to a parallel *quicksort* algorithm on a conventional, very well-balanced shared-memory multiprocessor (16 processor Sequent Balance system). The investigation begins with application of the UNIX profiling tool, gprof [GKM82]. TPT follows. After a determination of base performance has been made, TPT tunes the application, in one experimental step of 16 trials (runs), for a significant performance improvement--23% faster. The comparison reveals differences in profiling and TPT techniques and contrasts results from each. A second TPT iteration exemplifies possible stopping circumstances. The application under study is an implementation of C.A.R. Hoare's *quicksort* [H62]. It has been converted for a 16-processor Sequent Balance shared-memory system. 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 immaterial for purposes here. Tuning emphasizes improving what is available.

3.1.1 Conventional Micro-Level Measurement. Investigation begins by seeking information with the tool "gprof." Most UNIX and UNIX-based systems provide gprof to generate profiles of programs. Profiling tools help in debugging and in improving efficiency. A profile reveals which functions consume the most execution time and which are called most frequently. Once important functions are identified, their code can be improved. This paradigm has been successful for tuning sequential programs running on uniprocessors. However, on parallel architectures, emphasis changes. Considerations (usually non-existent in sequential programs) such as process interaction and processor idle time play a vital role in the performance of parallel programs. A simple extension of a sequential profiler on a multiprocessor can measure the sum of time a segment of code spent on each processor. This is one possible metric. However, a code segment's total processor time is not related in a simple way to parallel runtime (Anderson and Lazowska [AL90] 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. As will be clear shortly, TPT avoids many problems of having and interpreting metrics; it provides a better, more direct understanding of a code segment's impact on overall parallel performance.

Table I (below) gives abbreviated results from a standard gprof file for the *quicksort* routine. Two important metrics from the profile are the percentage of execution time consumed by a routine and the number of instances that routine was called. From the data it is clear that a majority of time for this parallel *quicksort* is spent in the s_lock() routine. In addition, this residency metric mildly suggests four other routines of concern, bubble_sort(), swap(), main(), and partition_list(). In terms of the number of times a routine was executed, only swap(), s_lock(), and s_unlock() stand out. From this information the programmer has to decide where to tune. Were the profile data for a sequential program on a uniprocessor, the task would be more obvious. Any time saved would be reflected, generally, in a shorter response.

Unfortunately, gprof data is murky in a parallel domain; interactions in the program structure and concealed wait states have to be accounted for (*e.g.*, [S91] show this). Perhaps the programmer will direct efforts to improve s_lock(), since it consumes the majority of overall runtime. Furthermore, sort(), swap(), partition_list(), and main() may be overlooked because improving these routines seems to offer little opportunity for performance improvement. Analyzing the data from the call graph perspective focuses attention upon swap(), s_lock(), and s_unlock(). Unfortunately (as with total execution time), it is again not easy to relate call count information to runtime effect in a parallel domain. In any case, a simple profile of a parallel program may not offer a clear plan of attack; alternatively, it could encourage a wrong interpretation of results.

Table I. Abbreviated quicksort results from gprof.

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

3.1.2 TPT's Macro-Level Measurements. TPT provides knowledge about chosen code segments (factors) indirectly by examining their combined effects over numerous performances. DEX analysis works backwards from response times and the pattern of treatment associated with each time. An additive treatment (e.g., a constant delay) is applied at locations (factors) in various code segments of interest. Those factors most sensitive to treatment (delay) signify segments that deserve first attention in tuning. The premise is that if a segment of code is highly sensitive to source code perturbations (i.e., delay has a clearly detrimental effect on performance), then source code improvements to that segment will have an opposite (positive) effect.

TPT testing of the parallel *quicksort* begins within the familiar framework of a 2^{6-2} fractional factorial experiment design [BHH78]. There are $2^4 = 16$ trials. Results from the analysis appear in Table II (following). 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. The first five factors are function calls. code1 is a segment of code (*i.e.*, a loop) within the main *quicksort* function: TPT can investigate code at resolutions higher than a function call. A plus sign in a given row denotes that treatment is set (*i.e.*, delay present). Minus denotes treatment absent (no time delay). Thus row 5 (− − + − + +) shows pop(), bubble_sort(), and 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 factor coefficients $\{\beta\}$. The sources column lists the most likely factors for an effect. Some confounding is deliberately introduced to shorten testing (see [BHH78]).

Table II. Calculated effects for 2**(6-2) factorial design, parallel quicksort example.

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
SE for the mean: S/2 = ± 0.05

It is clear from line 9 of Table II 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. The reasoning behind this is that s_lock() is highly associated with non-productive wait states. TPT 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* performance.

A second iteration of TPT demonstrates improvements in balance for 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 II-B, below, the reader should know that the dataset, synthetic delay function and experiment design differ from those used for Table II. This precludes direct comparison across the two tables without rescaling, which is provided in the scrap illustration following the discussion of Table II-B.

Table II-B. Effects for 2**(5-1) factorial design, improved version, parallel quicksort

mean and effects follow:

Source(s)	Mean 27.52 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
S.E. of the mean: S/2 = ±0.04

The largest effect in Table II-B is indicated by <--1. Unfortunately, it belongs to s_lock(), a system function that the programmer easily cannot change. The recourse is an algorithmic redesign that uses less of s_lock(). Although this would lie beyond what TPT

can recommend, TPT can certainly assist in the selection. The factor bubblesort(), shown as <--2, once 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. Among factors, one can recommend no single recoding improvement. To contrast the improved *quicksort*, denoted *qs+*, with the original, denoted *oqs*, main effects of Tables II and II-B have been normalized as percentages of unperturbed runs *for each sort version* (below).

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

3.1.3 Shared-Memory Results. TPT 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 [AL90], 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 I. 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), TPT is more straightforward. It simply avoids structural metrics and their interpretations. Comparing columns of results of gprof-like tools with TPT's results yields important distinctions. Numbers from gprof are raw measurements of interior (factor) detail, with which the user is to constitute models that predict a program's responses. TPT is exactly reversed. Program responses are first measured and then

correlated against precisely controlled changes in factors. Consequently, the TPT list of effects for factors is anything but raw data. Effects in the TPT model beckon to exactly those code locations with greatest impact upon performance. This is achieved with minor human effort, but multiple computer runs. The example shows TPT 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, TPT works as usual. This is largely because TPT, using macro-level analysis, does not rely upon precise internal system details.

The distributed-memory "Ring" example has been chosen because it exhibits a broader than usual set of TPT 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 3 (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 3. The outermost "B:loop" waits for a message (via a RECVW) from another node. This it passes 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. All 16 hypercube nodes have been programmed as a single ring; each Node Program B sends messages to one unique node and receives from another.

```
B: loop ...
     delay(F1); RECVW(); delay(F2)
     ...
     if(...)
          ....
          loop
             ..  {code A} ..
          endloop
          ...
          delay(F3); RECVW(); delay(F4)
     else ...
          delay(F5); RECVW(); delay(F6)
          ...
     SEND();
endloop: B
```

Figure 3. Node Program B--Slice w/Perturbations F1-F6.

TPT's 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 3, but appear in Table III as source labels for effects. Delay levels of treatment are 0 (rescaled as −1) and 10 (rescaled as +1). Delay location c that tests {code A} is thought important. Other segments are simply chosen uniformly throughout B. The experiment uses a 2^{8-4}, resolution IV design (see BHH78]), which 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 III, below). 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, as expected, the most sensitive by far (arrows, below).

Table III. Node Program w/Delays a-f

mean and effects follow:

		Mean 26.750000	
	Source(s)	Effect (Full)	
	a	-2.250000	..?
	b	-2.500000	..?
	ab+cg+dh+ef	2.750000	
-->	c	18.500000	<-- {code A}
	ac+bg+df+eh	2.250000	
	ag+bc+de+fh	2.500000	
	g	-2.750000	..?
	d	3.000000	
	ad+bh+cf+eg	-2.750000	
	ah+bd+ce+fg	-2.500000	
	h	2.750000	
	af+be+cd+gh	-2.500000	
	f	3.250000	
	e	3.000000	
	ae+bf+ch+dg	-2.250000	

Standard Error of an effect: S = ± 0.50
S.E. of the mean: S/2 = ±0.25

A further check of the importance of {code A} by varying available program parameters does reveal 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 added in the program code that all changes to {code A} should be done with care. This experience is similar to the earlier *quicksort* example. However, Table III hints there is more to be learned.

3.2.1 Optimization Rather Than Screening. The most sensitive factor in Table III, c, has an effect β_c that is 18.5/0.5 = 37 standard errors (deviations) removed from 0. It is thus extremely unlikely that (i) c's true effect is zero, and (ii) c's observed effect results from randomness inherent in the trials. 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 errors in magnitude, these special delay effects are worth investigating.

Delays a, b and c are near synchronous receives, RECVWs, in B. To investigate further, a second experiment is devised with a fresh copy of the program that has delays before and after all RECVWs. This is Figure 3 as shown, with 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 magnitudes for the higher delay setting uncover strong signs of nonlinearity in program response. More than two levels of delay seem appropriate.

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--see [KLF91]) uses only 25 trials in its search. Sample results follow.

Table IV. Complex Search of Delay Settings

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	

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 [J91], pp. 386-389, on "Informal Methods"). This straightforward approach can be quite effective, as one can see with the third line, above; trial 16 is unusual--for heavily communicating setups, certain *permanently installed* delays enhance B's speed--to twice as fast. The explanation is that heavily used message-passing provokes communication failures, which then generate retransmissions. Delays help synchronize nodes and cut 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 [F91]. 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 3 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, TPT indicates (i) which code is most sensitive to delays, and equally important, (ii) other sections that can be changed with impunity. {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 case (iii), the synthetic perturbations transcend their role in the investigation--they 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. Conclusions

Statistical screening is different from building a detailed performance model, *e.g.*, the FORTRAN virtual system in [SSM89]. TPT applies DEX screening to avoid the inefficiency of one-factor-at-time examinations as it searches for important factors. 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 [J91]. Examples are memory at 1 Mbyte/processor *versus* 4 Mbytes/processor, or file transfer via FTAM or FTP. But TPT 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 *ad hoc* substitution treatments whenever they labor under tightly constrained circumstances. Fortunately, TPT has more latitude, for real-time applications aside, source code is generally quite malleable. By deliberately avoiding component substitution, TPT 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. TPT avoids this problem by not depending upon detailed performance metrics. Similarly, interpretation of detailed metrics can be an uncertain process. Perturbation in some states may induce a change in response time that exceeds explanation via level-of-parallelism, the highest multiplier one might first expect [S91]. Interprocess communication makes some program states highly interdependent, with a combinatoric nature that strains analysis. TPT successfully addresses this problem by approaching the application and system as a complex, poorly understood entity. Structural interpretations within program or system then matter far less. Consequently, the technique is indifferent to the architecture of host system or test application.

Since it needs little internal detail, TPT 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 TPT opens many exciting new opportunities.) Analogous to its modest need of instrumentation, TPT in its basic form has a low visual demand. Simple character-display screens suffice. Tables II through IV are typical TPT displays.

Below are depicted some differences between conventional and TPT tuning paradigms:

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

4.1 Further Directions

While the two examples in the text illustrate essential aspects of TPT, the technique opens a rich field of possibilities and challenges. Consider the attribute of program size. The text examples are small, around 400 lines. But many applications are significantly larger, having written code of 1K to 10K source lines. The largest TPT 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 factors. For example, the statistical approach for large programs may shift away from focus upon factor interactions, which may be less probable. Instead, emphasis is given to screening large numbers of factors (64-128) in each iteration step.

Eventually, TPT 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.

4.2 Summary

TPT combines synthetic delays with DEX to yield an attractive new technique for MIMD program improvement. DEX lends to TPT some formidable powers of search and analysis, while in turn TPT'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.

5. Citations

[F91] J. Filliben. Private conversations with and notes to authors. November, 1991, NIST.

[J91] R. Jain. The Art of Computer Systems Performance Analysis. J. Wiley & Sons (New York, 1991), 685 pp.

[KLF91] 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.

[S91] R. Snelick. "Performance Evaluation of Hypercube Applications: Using a Global Clock and Time Dilation." NISTIR 4630, June, 1991, 26 pp.

[AL90] T.E. Anderson and E.D. Lazowska. "Quartz: A Tool for Tuning Parallel Program Performance." Proc., SIGMETRICS 1990 Conf., May 1990, 115-125.

[SSM89] R.H. Saavedra-Barrera, A.J. Smith, and E. Myia. "Machine Characterization Based on an Abstract High-Level Language Machine." IEEE Trans. on Computers 38, 12(Dec. 1989), 1659-1679.

[GKM82] 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.

[BHH78] G.E.P. Box, W.G. Hunter, and J.S. Hunter. STATISTICS FOR EXPERIMENTERS. John Wiley & Sons (New York, 1978).

[H62] C.A.R. Hoare. "Quicksort." Computer Journal 5, 1(January 1962), 10-15.

NIST-114A
(REV. 3-90)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY

BIBLIOGRAPHIC DATA SHEET

1. PUBLICATION OR REPORT NUMBER
NISTIR 4859

2. PERFORMING ORGANIZATION REPORT NUMBER

3. PUBLICATION DATE
JUNE 1992

4. TITLE AND SUBTITLE

Time-Perturbation Tuning of MIMD Programs

5. AUTHOR(S)

Gordon Lyon, Robert Snelick and Raghu Kacker

6. PERFORMING ORGANIZATION (IF JOINT OR OTHER THAN NIST, SEE INSTRUCTIONS)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
GAITHERSBURG, MD 20899

7. CONTRACT/GRANT NUMBER

8. TYPE OF REPORT AND PERIOD COVERED

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (STREET, CITY, STATE, ZIP)

Defense Advanced Research Projects Agency
3701 N. Fairfax Drive
Arlington, VA 22203-1714

10. SUPPLEMENTARY NOTES

11. ABSTRACT (A 200-WORD OR LESS FACTUAL SUMMARY OF MOST SIGNIFICANT INFORMATION. IF DOCUMENT INCLUDES A SIGNIFICANT BIBLIOGRAPHY OR LITERATURE SURVEY, MENTION IT HERE.)

Time-perturbation tuning--TPT--is a novel technique for assaying and improving the performance of programs on MIMD systems. Small, synthetic program delays are combined with statistically designed experiments (DEX). Claims for TPT are two: (i) Conceptually, it brings the powerful, mathematical perspective of experiment design to interdependent, sometimes refractory aspects of MIMD program tuning; (ii) Practically, it provides a needed speedup mechanism, synthetic time 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. TPT works on programs for both shared and distributed-memory, and it scales well with increasing system size.

12. KEY WORDS (6 TO 12 ENTRIES; ALPHABETICAL ORDER; CAPITALIZE ONLY PROPER NAMES; AND SEPARATE KEY WORDS BY SEMICOLONS)

code perturbation; designed experiments; factorial designs; MIMD; parallel programming; performance improvement; synthetic delays

13. AVAILABILITY

	UNLIMITED
X	FOR OFFICIAL DISTRIBUTION. DO NOT RELEASE TO NATIONAL TECHNICAL INFORMATION SERVICE (NTIS).
	ORDER FROM SUPERINTENDENT OF DOCUMENTS, U.S. GOVERNMENT PRINTING OFFICE, WASHINGTON, DC 20402.
	ORDER FROM NATIONAL TECHNICAL INFORMATION SERVICE (NTIS), SPRINGFIELD, VA 22161.

14. NUMBER OF PRINTED PAGES
27

15. PRICE
A03

ELECTRONIC FORM