# TRAPEDS:
# Producing Traces for Multicomputers Via Execution Driven Simulation

*Craig B. Stunkel and W. Kent Fuchs*

Computer Systems Group
Coordinated Science Laboratory
University of Illinois
1101 West Springfield
Urbana, Illinois 61801
stunkel@bach.csg.uiuc.edu (217) 244-7178
fuchs@bach.csg.uiuc.edu (217) 333-9731

Address communications to: *Craig B. Stunkel*

## ABSTRACT

Trace-driven simulation is an important aid in performance analysis of computer systems. Capturing address traces for these simulations is a difficult problem for single processors and particularly for multicomputers. Even when existing trace methods can be used on multicomputers, the amount of collected data typically grows with the number of processors, so I/O and trace storage costs increase. A new technique is presented in this paper which modifies the executable code to dynamically collect the address trace from the user code and analyzes this trace during the execution of the program. This method helps resolve the I/O and storage problems and facilitates parallel analysis of the address trace. If a trace stored on disk is desired, the generated trace information can also be written to files during execution, with a resultant drop in program execution speed. An initial implementation on the Intel iPSC/2 hypercube multicomputer is detailed, and sample simulation results are presented. The effect of this trace collection method on execution time is illustrated.

*Acknowledgment:* This research was supported in part by a Shell Doctoral Fellowship, by a Digital Faculty Incentives for Excellence Award, and by 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).

## 1. INTRODUCTION

Trace-driven simulation is an important method of analyzing the performance of computer systems [1,2]. However, accurately and efficiently capturing address trace data for multicomputers is extremely difficult. In this paper, we examine the problem of address trace generation and collection for *multicomputers*, which are non-shared distributed memory parallel processors of the multiple-instruction, multiple-data stream class (MIMD) [3]. This class of machines is in contrast to MIMD *multiprocessors* with a global shared memory. Recording the address traces for multicomputers typically requires large amounts of memory, and therefore the I/O necessary for saving these traces is a significant overhead. In addition, the traces gathered are typically valid for only the number of processing nodes that participated in the execution.

Understanding how the execution time, speedup, and other system measures change as the number of processors change is of vital importance to multicomputer hardware, software, and application designers. This necessity mandates having several sets of traces for any single application problem - one set of traces for each possible dimension hypercube, for example. Also, since speedup is heavily dependent on the size and characteristics of the input data for most parallel applications, there is a need for application program traces for several different sets of inputs. Keeping all of these traces in storage rapidly becomes an impracticality for a large number of processing nodes.

This paper presents a new software address tracing technique for multicomputers called TRAPEDS - TRAce-Producing Execution Driven Simulation. This software technique modifies executable code (at the assembly language level), producing a new executable program which dynamically produces correct address traces of the user code and other information valuable in assessing computer system performance. The primary purpose of this tool is to enable hardware designers to model and simulate trade-offs in a multicomputer's computation and communication capabilities for the specific parallel algorithms that are traced.

As trace addresses are generated by the execution, analysis of cache performance and other design alternatives can be immediately performed, eliminating the need for storing large amounts of trace data, and thereby reducing the massive trace storage requirement and the I/O bottleneck that would slow execution on a multicomputer. An added benefit of collecting *and* analyzing the address trace on the multicomputer is the

speedup obtained as the number of processors increases. The simulation speedup is dependent on the speedup of the original executable code, since the effects of synchronization, message-passing, and unbalanced or replicated computation are also present in the modified executable code. Our approach to producing address trace data has been implemented on an Intel iPSC/2 hypercube multicomputer. In our implementation on the iPSC/2, the execution of the program has been degraded by less than a factor of 50, which compares favorably with existing trace collection methods. Conventional stored trace data can also be obtained with the TRAPEDS approach with the resulting increase in storage cost and performance degradation.

A brief review of popular existing trace methods is presented in §2. The TRAPEDS methodology is discussed in §3, and its implementation on the iPSC/2 is outlined in §4. §5 presents results of performance evaluation of this method along with preliminary memory reference observations.

## 2. REVIEW OF EXISTING TRACING TECHNIQUES

### 2.1. Hardware Monitoring Based Traces

Hardware monitoring can directly record memory bus activity and the actual addresses sent to off-chip caches or main memory modules. This monitoring captures both user and operating system references, as well as multiprogrammed streams of references. The effect of on-chip caches on the reference stream is also included, but this *implementation* effect is also a drawback of hardware monitoring. The primary limitations of this approach are its complexity, cost, and lack of easy flexibility. Because of limited memory and bandwidth, hardware monitors typically cannot capture all of the reference trace, and must settle for isolated collections of contiguous references, or counts of events, rather than a listing of the events themselves. To collect trace information for all of the processors in a multicomputer, the complex hardware required grows at least linearly with the number of processors.

### 2.2. Instruction Interrupt Based Traces

Some computer systems provide the capability of *interrupting* the execution of a program after each instruction. The virtual address references for each type of instruction can then be calculated. Since operating system routines typically disable these interrupts, the operating system execution cannot be traced. The need to

interrupt each instruction slows down the program execution considerably. For multicomputers, this distortion of instruction execution time inevitably changes the fashion in which different processing nodes interact with each other (except when the multicomputer message-passing is synchronized between a specified sender and receiver, such as in the Occam language [4] ), and thereby possibly changing the address trace.

### 2.3. Software Simulation Based Traces

Software simulation can also provide accurate user traces, and can simultaneously model the execution time of a processor, which can enable accurate modeling of the interaction between different processors in multicomputers. This simulation can also provide emulation of operating system activities, although this emulation may not be exact. Software simulation is slow, however, since the simulator must model much of the real hardware, including the actual ALU operations, flag setting, instruction fetching, and main memory storage and accesses [5,6].

### 2.4. Microprogramming Based Traces

ATUM [7] is a recently introduced technique that alters a machine's microcode to capture address traces. This technique enables the capture of full address traces for multiprogrammed user code and operating system activity. It is also fast, with factor of 20 overhead reported. This technique was recently used to collect traces for a 4-processor system [8]. Despite the significance of this approach, there are obstacles to implementing microcode alteration on existing multicomputers. The main obstacle is that the processors on commercial multicomputers tend to be one-chip microprocessors, and either do not use microcode or contain their microcode in ROM (as in the iPSC/2, which uses the 80386 processor). Even if this microcode could be changed, there would typically not be extra space on the chip to allow the ATUM changes.

### 2.5. TRAPEDS Based Traces

The TRAPEDS method of this paper addresses the issues of producing accurate and efficient multicomputer traces with a reduction in the burdensome storage and I/O requirements of stored traces. The traces produced by this method do not include operating system references. Also, the current implementation on the iPSC/2 does not provide the ability to collect multiprogrammed traces. At the present time, multicomputers such as hypercubes are

rarely used in a multiprogramming mode, partly because each processing node has a fixed amount of space into which all currently executing programs must completely reside. TRAPEDS also attempts to mitigate the effects of execution time distortion on the interaction between processors by introducing a simulated time for each multicomputer processing node, and by passing these simulated times between nodes during communication.

## 3. TRACE-PRODUCING EXECUTION-DRIVEN SIMULATION METHOD

Execution-driven simulation is a term coined by Covington, et al. [9], for an approach to gathering accurate timing statistics for a program as it is executing. Briefly, the method estimates the time to execute each *basic block* in the assembly code, where a basic block is defined as a set of machine instructions that will always execute together in the absence of interrupts. Calls to a simulation timer update routine are placed at the beginning of each basic block in the assembly code, and the estimated execution time for that basic block is passed as a parameter to this routine. The execution of this modified program also updates the timer, simulating the program execution time. This method was also used by Fujimoto [10], and in the instruction counting method introduced by Weinberger [11].

The execution driven simulation approach can be easily extended to perform a static address analysis on each basic block. Static analysis can generate instruction address traces, but data addresses cannot be fully determined until execution, so information must be collected and analyzed dynamically during program execution to produce full user address traces. Our paper extends the execution-driven simulation to enable full user address tracing for both instructions and data.

The dynamic collection of information utilized in this paper requires additional modification of the assembly code. In addition, static analysis produces address information that must be stored in the virtual address space. For these reasons, addresses collected during execution may not be identical to the actual addresses in the unmodified code. Calculating the *correct* data addresses at execution time, therefore, is a major element of the TRAPEDS method. The steps used to produce a modified executable program are described in what follows. All steps are accomplished automatically by the TRAPEDS software.

### 3.1. TRAPEDS Steps in Modifying the Executable File

**STEP 1:**

The original program's source files are compiled and linked with the library functions to produce the original executable file as is illustrated in Figure 1. This file is analyzed to record the beginning virtual addresses of the text (program), initialized data, and uninitialized data sections.

**STEP 2:**

All source written in C is compiled to assembly language. Together with any source files written directly in assembly language, these compiled programs form the suite of assembly language files that will be modified by the TRAPEDS software.

**STEP 3:**

For each resulting assembly language file, the corresponding machine language instructions in the executable file are analyzed. Utilizing both the assembly language and machine code is advantageous because extracting virtual address information requires the actual machine language instructions. However, it is far easier to modify the associated assembly language program to capture necessary run time address information.

**STEP 4:**

The assembly source is broken into basic blocks by noting labels and statements such as jumps, calls, and returns that can break the normal sequential execution of the program. In a separate assembly language file (named `auxfile.s` in this discussion), the starting address of the basic block is recorded, the first of several types of data that will be recorded in `auxfile.s` for each basic block. A call to the basic block performance simulation routine (hereafter called `X_bb_perf`) is inserted at the beginning of each basic block in the assembly source file. A pointer to the `auxfile.s` address information is also saved in a global variable before this call to `X_bb_perf`. Note that since the dynamic address information is collected *during* the execution of a basic block, the call to `X_bb_perf` must analyze the *previously* executed basic block. This is conceptualized in Figure 2, which shows the high-level organization of `X_bb_perf` in a C-like syntax.

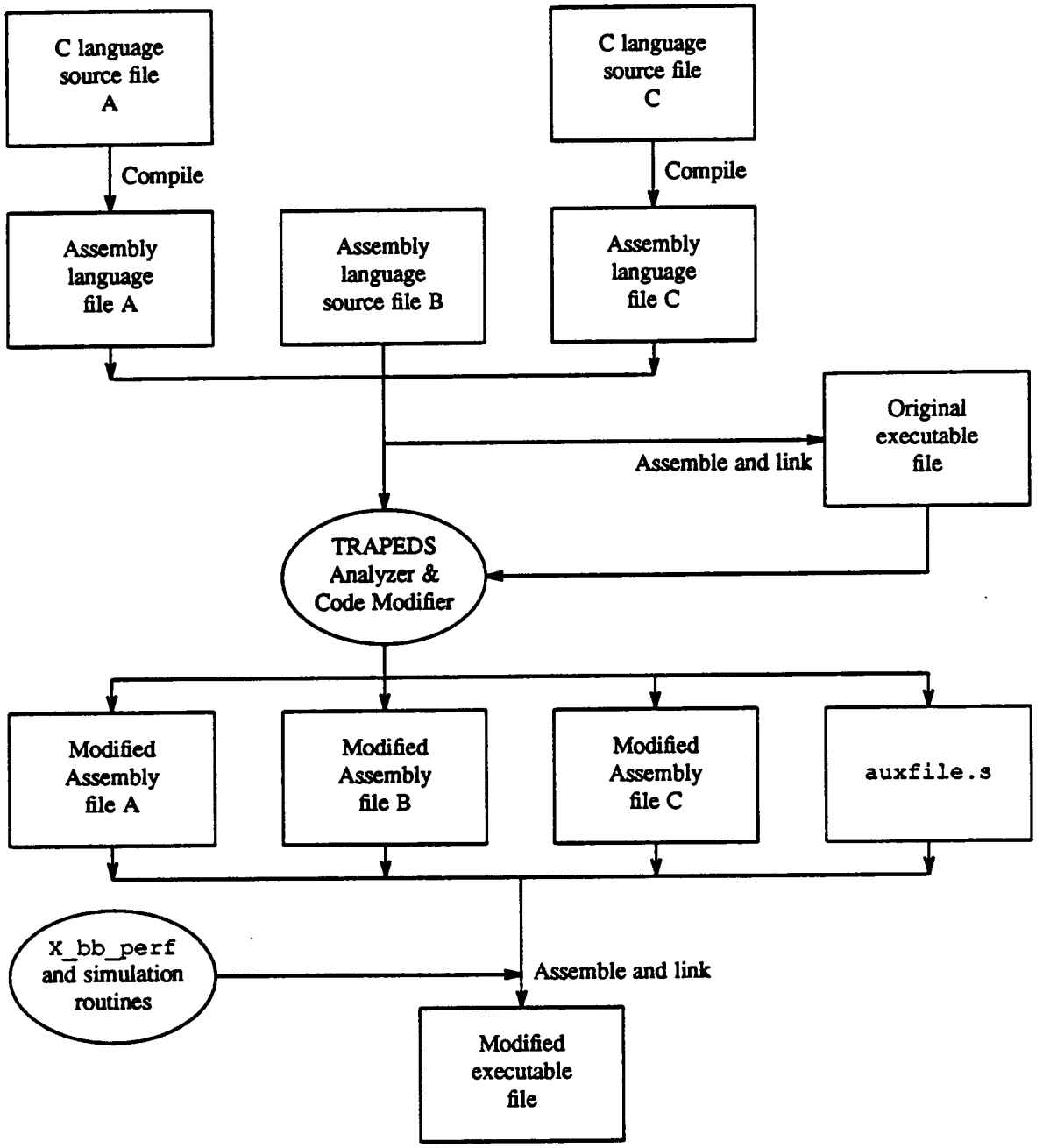


**Figure 1. TRAPEDS steps for creating modified executable files.**
**(shown analyzing three source files - two C files and one assembly file)**

```
long  X_bb_pointer,  X_last_bb_pointer;

X_bb_perf ()
{
        long  auxfile_array_address = X_last_bb_pointer;

        /* calculate addresses using information in auxfile.s
           and run time information saved by the modified assembly file */

        /* call cache simulation routine */

        /* update simulated execution time */

        X_last_bb_pointer = X_bb_pointer; /* X_bb_pointer is the global
                                             variable saved before each
                                             call to X_bb_perf()  */
}
```

**Figure 2. High-level structure of `X_bb_perf`, the basic block performance analysis routine.**

**STEP 5:**

For each instruction that accesses memory, the type of access and the addressing mode of its memory references are recorded in `auxfile.s`. Also recorded are the number of instruction fetches required to load the instructions executed since the previous memory reference.

**STEP 6:**

For each memory reference, the calculation of the virtual address may involve *static* address information such as address displacements, *dynamic* address information such as a base and/or index register values, or a combination of static and dynamic values. Any static information is saved after the type of access in `auxfile.s`.

**STEP 7:**

For each dynamic part of the address, instructions are inserted into the assembly code to save their values at run time by moving them to a reserved area of global memory. After processing by `X_bb_perf`, these values can be discarded. Hence, this reserved memory area need only be large enough to store the largest number of dynamic address values needed in any given basic block of the executable file.

**STEP 8:**

The modified assembly files are assembled again, and linked with `X_bb_perf` and any simulation routines

called by `X_bb_perf`, resulting in a modified executable file capable of generating address trace information. The problem mentioned earlier involving changed virtual addresses still exists at this point, however.

### 3.2. Solving the virtual address modification problem

The discussion in this section is based on UNIX[1] System V, but the principles considered apply to other UNIX operating systems, and many other operating systems as well. In UNIX System V, an executable file is commonly divided into three segments - `.text`, `.data`, and `.bss` [12]. The `.bss` segment (the stack) starts at virtual address 0, and the `.text` and `.data` segments start at an identical virtual address (usually address 0, but not necessarily). The `.text` *section* within the `.text` segment contains user code. The `.data` segment has two adjacent sections - the first section contains *initialized* internal static data and external data,[2] which we shall refer to as initialized *permanent* data, because the location of the data is reserved during the entire execution of the program. The second section contains *uninitialized* permanent data. The `.bss` segment contains no initial data, and merely indicates that a stack segment is required. The `.text` and `.data` segments and sections are pictured in Figure 3(a).

The initialized `.data` section starts in the next page table directory after the last one used by the text section, in the first page of that directory, with an offset into that page equal to the first unused memory offset in the last page of text. This allows the `.text` and `.data` sections, which are physically adjacent in the executable file, to be loaded adjacently into physical memory. This also implies that any changes in the size of the `.text` section will change the starting virtual address of the `.data` section. In addition, the extra permanent data in `auxfile.s`, `X_bb_perf`, and the performance analysis routines called by `X_bb_perf` change the virtual addresses in the `.data` sections.

In TRAPEDS, the solution involves ensuring that all newly created permanent data are *initialized* and placed at the beginning of the initialized `.data` section, as illustrated in Figure 3(b). In this case, all the original data in

[1]UNIX is a registered trademark of AT&T.

[2]C language terminology is used here. In C external data corresponds to variables defined outside of any function. Internal static variables are defined inside a routine but retain their values between successive invocations of the routine [13]. All other variables are allocated space on the stack when their routine, or sometimes even a subset of statements within a routine, is executed, and disappear after the routine is finished.

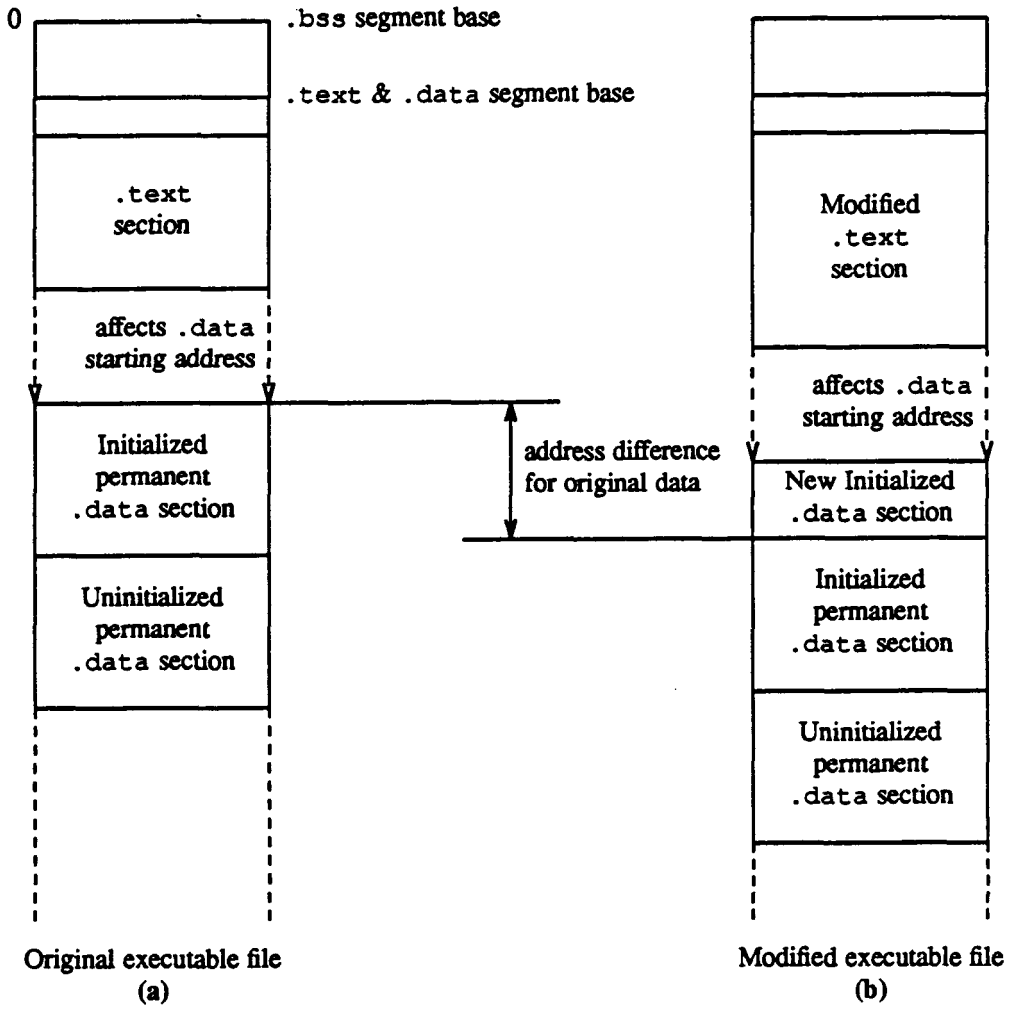


**Figure 3. Placement of code and data in the virtual address space of the executable files.**

the `.data` sections will be displaced by an equivalent amount. This displacement is a result of both the `.text` and `.data` section changes, and can be easily determined by comparing symbol table information in the modified and original executable files. Another requirement is that either static or dynamic analysis should be easily able to determine which segment the referenced data is in, because `.bss` segment addressing remains unaffected by the changes in the `.text` and `.data` sections (no address adjustment is needed).

One subtle problem remaining is that `X_bb_perf` and its simulation routines directly or indirectly call several library routines, some of which may define initialized or uninitialized permanent data. If these routines were already called in the original executable file, calling them in `X_bb_perf` could cause their associated

permanent data could be placed into the `.data` sections in a different order in the original executable file. This problem is overcome by a dummy assembly language routine that calls each routine in the same order as they are found in the original executable file. This dummy assembly language routine is linked at the start of the `.text` section and must never be called.

A second problem arises when `X_bb_perf` or its simulation routines directly or indirectly call library routines with permanent data that were *not* called by the original source files. In practice most library routines do not define permanent data, and this problem does not exist with our current performance routines. The solution to this problem involves linking the previously uncalled library routines to `X_bb_perf` and its routines during a first-pass linking phase. If the permanent data in these library routines is uninitialized (very rare), this data must be initialized. With this procedure, all new permanent data will be placed before the original permanent data by the normal linking of all routines.

## 4. IMPLEMENTATION ON THE 80386-BASED IPSC/2

The Intel iPSC/2 hypercube is an 80386/80387-based multicomputer that can contain up to 128 processing nodes, each with up to 16 Megabytes of main memory. The TRAPEDS method was implemented for a 16 node iPSC/2 with 4 Megabytes of main memory at each node. Each processor also has a 64 Kbyte zero wait-state write-through cache with a 4 byte line size and direct mapping. The 80386 pre-fetches instructions into a 16 byte buffer via its 4 byte data bus [14].

On the 80386, all explicit references to memory use the same addressing modes for the segment offset, which are subsets of the following general addressing mode:

```
[base-register] + (index-register * scale-factor) + displacement
```

The displacement and scale factor, if present, constitute static information saved in `auxfile.s` during the assembly code modification. The base register and index register, if present, constitute dynamic information that must be saved at execution time. When only one program is running on a hypercube node, the `.bss`, `.text`, and `.data` segments all start at address 0. Hence, the segment offset is equal to the virtual address.

As shown before, this virtual address may not be correct. In the 80386, the instruction implicitly or explicitly indicates the segment used for memory references. Any references to the `.data` segment that also use a base register have incorrect (modified) addresses, and the correcting offset is subtracted from the calculated address for these cases.

`X_bb_perf` recognizes several types of memory accesses, such as push, push memory, pop, pop memory, read memory, read and write memory, write memory, read 2 words of memory, etc.. The segment (usually `.data` or `.bss`) referenced is also stored as part of the access type. Combined with the addressing mode, these types of accesses provide a full description of every memory access.

The current implementation records the type of access and addressing mode in `auxfile.s` as shown in Figure 4. If the recorded addressing mode contains a displacement, this displacement is placed in the 4 bytes following the mode information. The `auxfile.s` information for each basic block also contains the starting text address of the basic block, and along with the code fetching information saved in bits 16-23, allows `X_bb_perf` to fully reconstruct and interleave the code and data accesses to form an accurate trace.

In collecting traces on a multicomputer it is important to model the interaction between processors as accurately as possible. The trace collection slows down the execution of each program and thus potentially changes the order in which processors send messages. In this implementation, one of the functions of `X_bb_perf` is to simulate the elapsed number of cycles in each processor's execution. This number of cycles is stored in an 8 byte field `X_time`, since using only 4 bytes to count cycles would cause wrap-around of the time to zero in less than 5 minutes of simulated execution time of the 16-MHz 80386 processor. The information stored in `auxfile.s` for each basic block also contains the estimated number of processor cycles for that basic

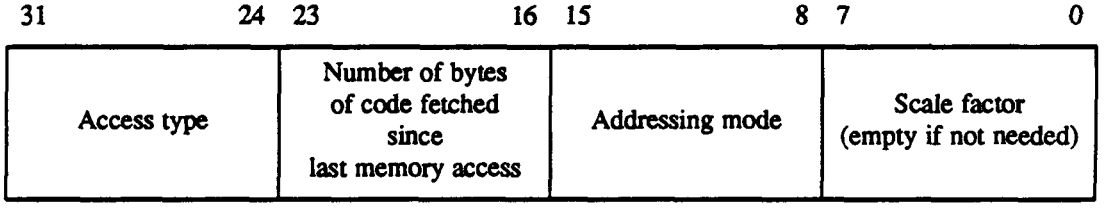


**Figure 4. Structure of the type of access and addressing mode information in `auxfile.s`.**

block (assuming no cache misses), and `X_time` is incremented by that amount when the basic block is executed. The effects of cache misses are modeled simplistically by adding a fixed number of cycles penalty for each type of memory access. The 80386 also contains a 32 entry 4 way associative TLB, but the TLB and the effects of TLB misses are *not* modeled in the current implementation.

The iPSC/2 message-passing routines have a top layer of code implemented in C that calls the actual operating system code. This top layer of code was provided to allow simple modifica tionsof message passing. The TRAPEDS simulation routines contain redefined message sends that send an extra message containing `X_time` for every normal message send. The receive routines are recoded to receive `X_time` after every normal receive. In this manner communication can be synchronized to model the actual execution, even though the modified executions may be interacting differently. Each modified send and receive routine models the cost of communication (both latency and per byte transmission speed) in a simple manner which does *not* account for possible delays in message routing caused by network congestion, two or more messages arriving at a node in the same time interval, etc. Thus the modeled communication contains several inaccuracies. However, synchronizing the processors through the simulated `X_time` values is more likely to produce accurate traces because an attempt is made to counter the effects of execution overhead. Another purpose of `X_time` is modeling the performance of the iPSC/2 hypercube as various hardware parameters are changed. Graphs of simulated speedup will be compared to actual speedup in §5.

## 5. TRAPEDS PERFORMANCE AND CACHE SIMULATION RESULTS

This section discusses TRAPEDS simulation performance and data collected by the TRAPEDS method. Benchmark studies of the overhead of this method are particularly emphasized, since the modified executable file requires more memory space and more execution time than the original executable file.

### 5.1. Space and Time Overhead

Both the `.text` section and the initialized permanent `.data` section are substantially lengthened by additional information. This amount of additional information increases with the number of basic blocks and memory references in the original `.text` section. The cache model is part of the additional `.data` section, and

is listed separately because it is not dependent on the size of the original `.text` section. Table 1 shows the difference between the original and modified section sizes for three iPSC/2 hypercube node programs. Much of the additional data overhead is for a cache memory model array which is included in the simulation routines, and is shown large enough to store up to 16K tags for a direct mapped cache. The dependence of `.text` and `.data` section overhead on original `.text` section size is also apparent. When compared to the total iPSC/2 node memory space (4 Megabytes), the space overheads are quite small.

Extra execution time is incurred saving register values and calling `X_bb_perf` at the start of each basic block. Extra overhead is also incurred for any simulation routines called by `X_bb_perf`. It is desirable to separate the effects of these two overheads, since for a given program and hypercube dimension the overhead due to `X_bb_perf` execution should be relatively constant, while the simulation routines can be changed for each new run (e.g., changing from a direct mapped cache to a set-associative cache model or simulating two or more cache models in the same run will cause an increase in the cache model simulation time). The plots to be shown for execution overhead assume the following definition:

$$execution\ overhead = \frac{modified\ file\ execution\ time}{original\ file\ execution\ time}$$

To separate the effects of address generation from simulation, performance benchmarks were run against a parallel version of the simplex algorithm [15]. This algorithm has moderate but not excessive parallelism, and this parallelism is sensitive to changes in the input data size, allowing some control over algorithm speedup. The complexity of sequential simplex is roughly proportional to $m^2n$, where $m$ is the number of rows in the input matrix, and $n$ is the number of columns. For each graph shown, the the number of rows and columns in the input

| Program Function | Original `.text` bytes | Additional `.text` bytes | Additional `.data` bytes | Additional cache model bytes | Total additional bytes |
|---|---|---|---|---|---|
| Gaussian Elimination | 12196 | 6764 | 5172 | 65536 | 77472 |
| FFT | 15273 | 8194 | 7496 | 65536 | 81226 |
| Simplex Algorithm | 18780 | 11272 | 9832 | 65536 | 86640 |

**Table 1. Space overhead for modified executable files (in number of bytes).**

are displayed.

In the first performance experiment `X_bb_perf` generates addresses and calls the cache simulation routine, but the simulation routine immediately returns to `X_bb_perf`. A plot of the execution overhead for the original and modified executable files is shown in Figure 5. Execution overhead due to TRAPEDS is ≈ 30 for execution on a single node. As the number of nodes increases, however, the overhead factor decreases. This is a consequence of the parallel nature of the address tracing overhead.

Because the overhead is decreasing with the hypercube dimension, the speedup $S(mod)_n$ of the modified executable file is higher than the speedup $S_n$ for the original file, where these speedups are defined as:

$$S(mod)_n = \frac{single\ node\ modified\ file\ execution\ time}{modified\ file\ execution\ time}$$

$$S_n = \frac{single\ node\ original\ file\ execution\ time}{original\ file\ execution\ time}$$

Figure 6 illustrates this effect for two different sets of input data. The positive effect on $S(mod)_n$ decreases as the parallelism of the original program increases.

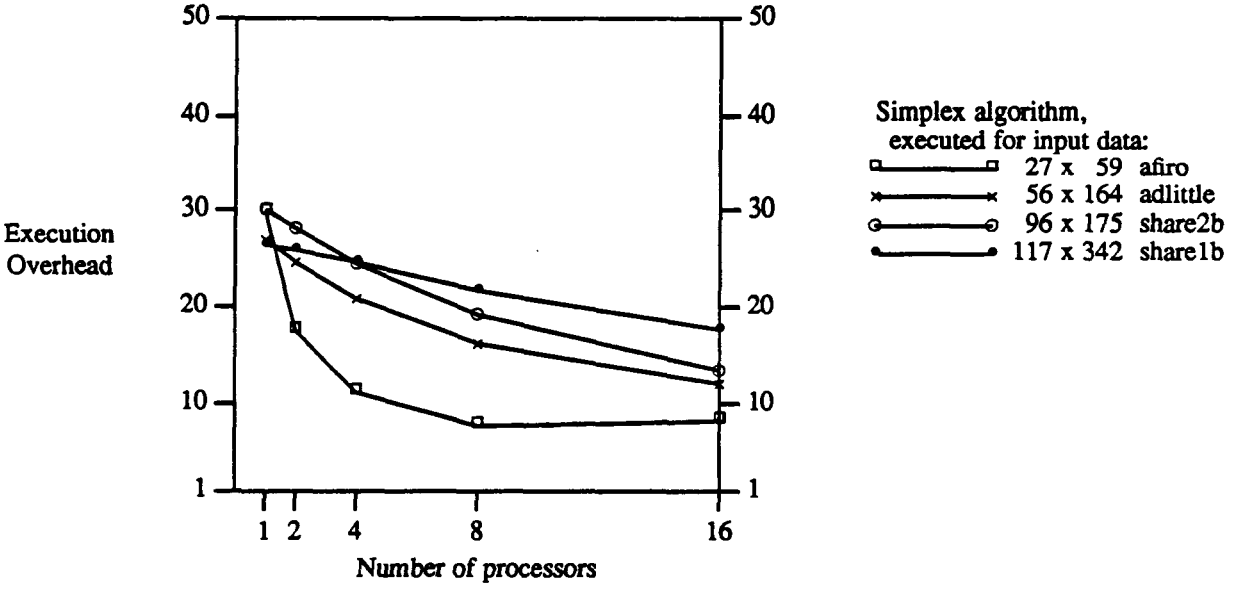


**Figure 5. Execution overhead for trace address generation only.**

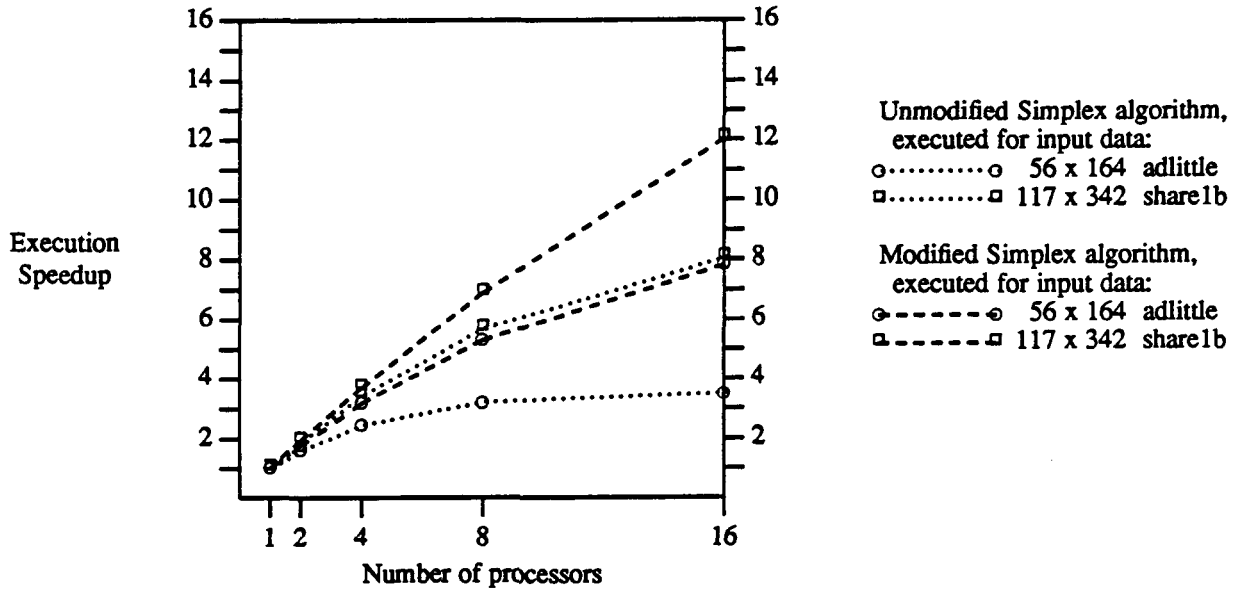


**Figure 6. Speedup for generating trace addresses only.**

The second experiment shows the performance overhead when a direct mapped cache simulation model is used to find cache hit ratios. Figure 7 shows this overhead for different sizes of hypercubes. For a program executing on a single hypercube node, the execution overhead for producing trace addresses and simulating cache hits and misses for a *direct* mapped cache is less than a factor of 50. Again, increasing the number of hypercube nodes decreases the execution overhead, this time even more dramatically, which implies improved speedup $S(mod)_n$.

Even for the simple direct mapped cache model used, the cache simulation constitutes a significant portion of the total execution overhead. Additional complexity in the cache model or simulating cache performance for two or more caches will increase the execution overhead further, but the resulting analysis will be conducted with an even higher degree of parallelism.

Also of interest is the execution overhead when traces are being saved to disk. For this experiment, each hypercube node was assigned a specific disk output file for trace address storage. On the iPSC/2, all communication to the host machine (which is the only processor with direct access to the disk) must pass through a single hypercube node, called node 0. Thus node 0 and the host machine provide a substantial I/O bottleneck as

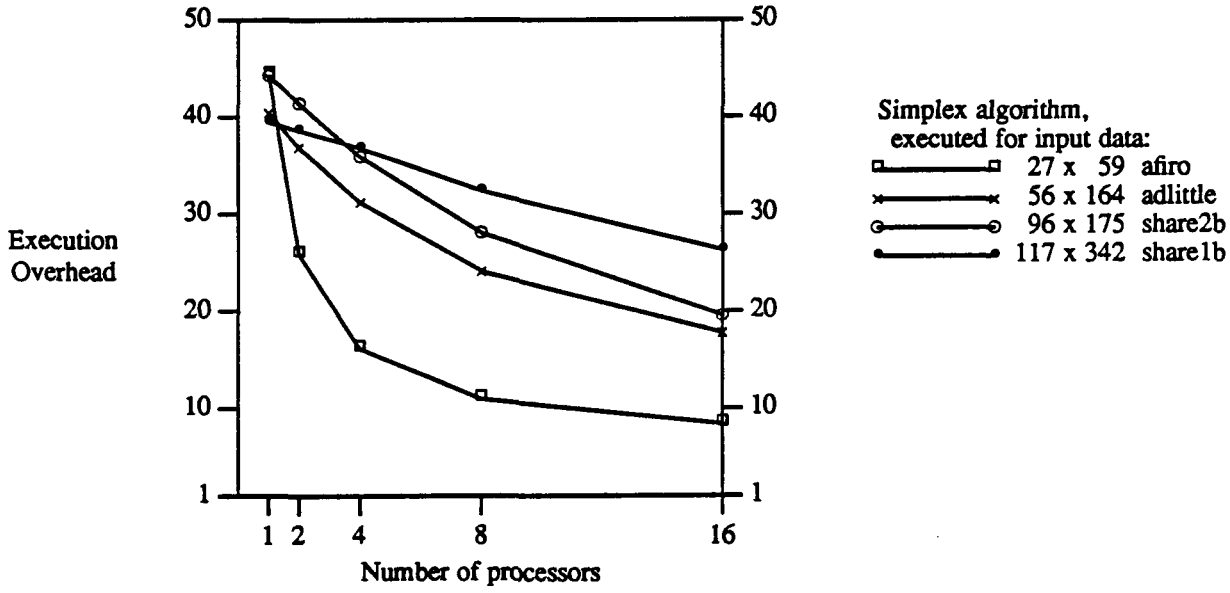


**Figure 7. Execution overhead for trace address generation and direct mapped cache simulation.**

the number of nodes increase. The disk space on our system was limited, so we simulated the writing of these files by periodically reverting to the beginning of the file before writing the trace information. This procedure still requires the same node communication as is required for full trace storage, thus the location of disk writes are the only factor altered. Figure 8 shows the results of this experiment.

It is obvious that storing the traces from the iPSC/2 is very costly, and this cost increases substantially as the number of nodes in the hypercube increases. The size of a block of addresses influences this overhead. Larger block sizes appear to be inefficient for a small number of nodes, but become more attractive as the number of nodes in the iPSC/2 increases. For 16 nodes, saving the traces addresses to disk is about 2 orders of magnitude slower than generating and analyzing these addresses concurrently. In general, it makes little sense to store the multicomputer traces if the multicomputer is available for use in trace analysis, and if the extra analysis routines do not cause the execution module's size to grow beyond the memory limit of a multicomputer node.

### 5.2. Cache Data From Multicomputer Traces

Although the purpose of this paper is not the analysis of multicomputer cache performance, a small sample of memory access and cache hit ratio data will be presented to discuss some multicomputer issues different than

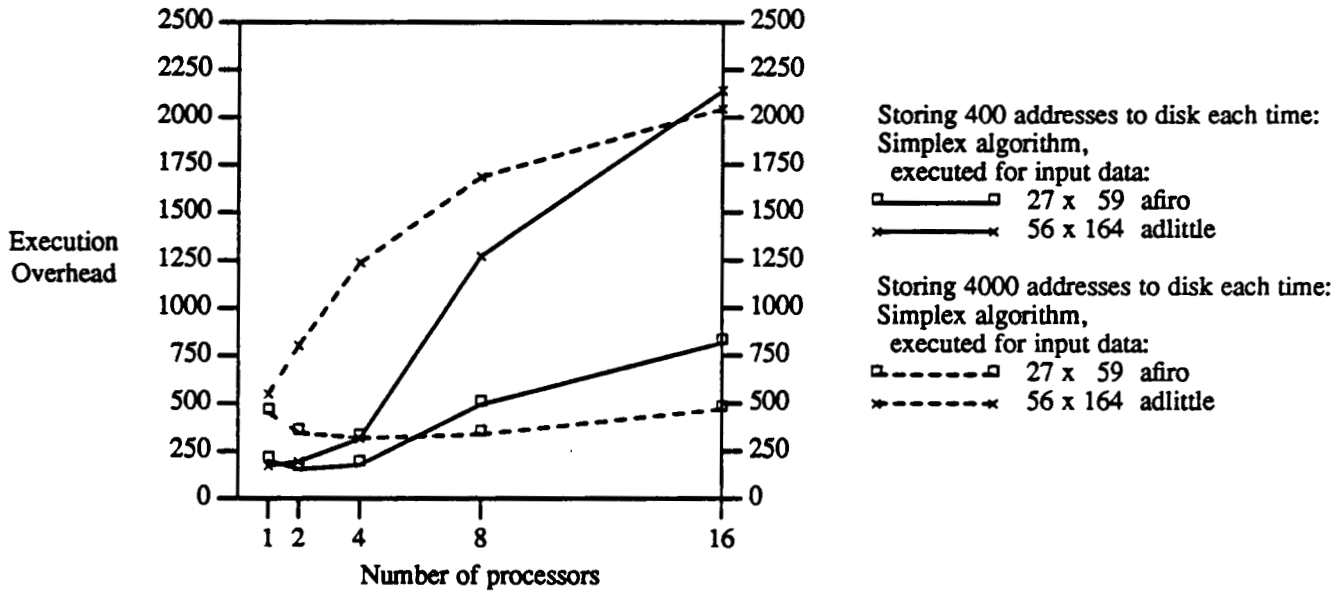


**Figure 8. Execution overhead for generating addresses and storing them to disk files.**

issues found in single processor studies. One such issue is the variance of number of memory accesses and cache hits among different nodes of a multicomputer. Another example issue is how the number of memory accesses and the cache hit ratio change with the number of nodes for a given problem.

To illustrate variance in the number of memory accesses, Figure 9 shows the total number of text reads, data writes, and data reads for the simplex algorithm on a 4 node hypercube. This data was captured by `X_bb_perf` as it generated the addresses. Given the difficulty of collecting traces via existing methods, it is tempting to capture an address trace from a single node and assume it is representative. For a variety of obvious reasons - unbalanced workload, unbalanced communication and synchronization requirements, etc., this will not always be true.

Figure 10 shows the dependence of cache hit ratio on the number of nodes for the simplex algorithm with a 64 Kbyte direct mapped cache with a 4 byte line size. This data was captured by a cache model called by `X_bb_perf` after it generated each address. For the input datasets shown, the highest hit ratio is always found for the single processor case. This indicates that most of the code and data for these problems fits well in the 64 Kbyte cache. As the number of processors increases, less total work is done by each processor, taking less

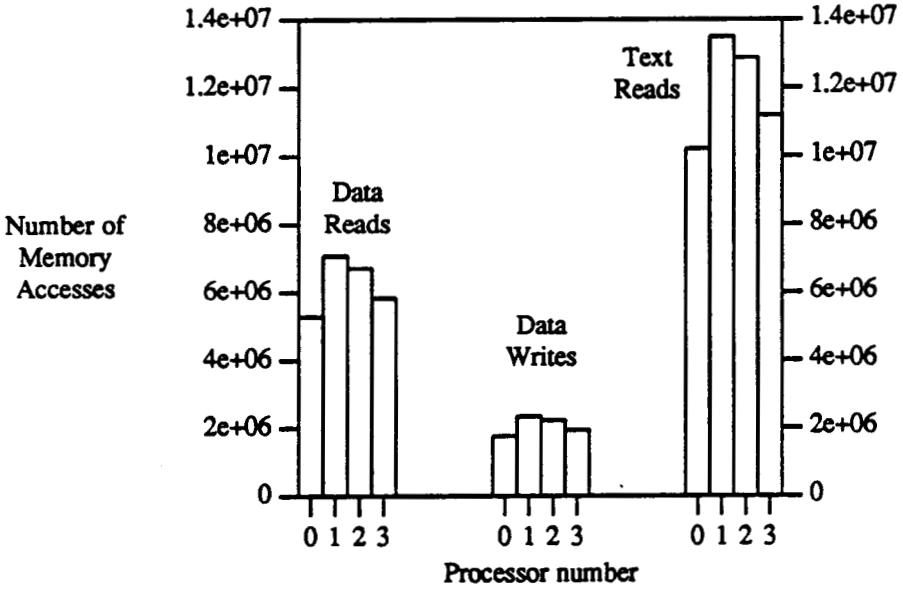


**Figure 9. Total number of text reads, data reads, and data writes for a 4 node hypercube execution.**

advantage of the code and data that already resides in the cache. It is likely that a larger number of nodes will yield a better cache hit ratio for problems with large amounts of data that can be partitioned well among the nodes.

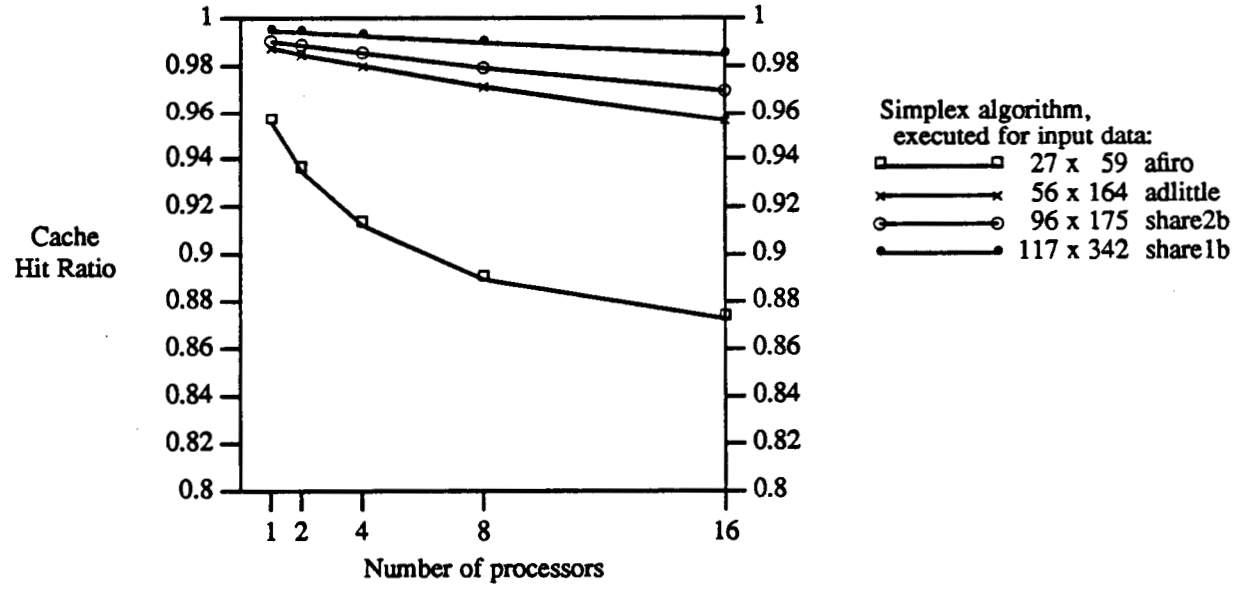


**Figure 10. Effect of number of nodes on the cache hit ratio for a 64 Kbyte direct mapped cache, 4 byte line size.**

## 6. CONCLUSIONS

This paper presents TRAPEDS, a new method of producing address traces. This method analyzes the program at the assembly language level to create modified executable files that produce the address traces. The modified executable files run less than a factor of 50 slower than the original executable files, which compares favorably with existing software trace gathering approaches. Benchmark studies show that the execution overhead of the TRAPEDS method decreases as the number of processors traced increases.

Drawbacks of the TRAPEDS approach include no traces of operating system addresses, no current ability to collect multiprogrammed traces, and slower execution than with hardware trace capture. Only virtual addresses of user code are captured with TRAPEDS.

The TRAPEDS method has particular advantages for multicomputer systems. The problems of I/O and storage for trace generation and trace usage for multicomputers are resolved by analyzing in parallel the generated addresses during the collection process.

## REFERENCES


[1] P. Heidelberger and S. S. Lavenberg, "Computer Performance Evaluation Methodology," *IEEE Trans. on Computers*, vol. C-33, pp. 1195-1220, December 1984.

[2] A. J. Smith, "Cache Evaluation and the Impact of Workload Choice," *Proc. 12th Ann. Int. Symp. on Computer Architecture*, pp. 64-73, June 1985.

[3] M. J. Quinn, *Designing Efficient Algorithms for Parallel Computers*. New York: McGraw-Hill, 1987.

[4] INMOS Limited, *Occam 2 Reference Manual*. New York: Prentice Hall, 1988.

[5] T. S. Axerold, P. Dubois, and P. Elgroth, "A Simulator for MIMD Performance Prediction - Application to the S-1 MkIIa Multiprocessor," *Parallel Computing*, vol. 1, pp. 237-274, 1984.

[6] K. So, F. Darema-Rogers, D. A. George, V .A. Norton, and G. F. Pfister, *PSIMUL - A System for Parallel Execution of Parallel Programs*. Yorktown Heights, NY: IBM T. J. Watson Research Center.

[7] A. Agarwal, R. L. Sites, and M. Horowitz, "ATUM: A New Technique for Capturing Address Traces Using Microcode," *Proc. 13th Ann. Int. Symp. on Computer Architecture*, pp. 119-127, June 1986.

[8] R. L. Sites and A. Agarwal, "Multiprocessor Cache Analysis Using ATUM," *Proc. 15th Ann. Int. Symp. on Computer Architecture*, pp. 186-195, May 1988.

[9] R. C. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair, "The Rice Parallel Processing Testbed," *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pp. 4-11, May 1988.

[10] R. M. Fujimoto, "SIMON: A Simulator of Multicomputer Networks," Report No. UCB/CSD 83/140, Computer Science Division (EECS), Univ. of California, Berkeley, September 1983.

[11] P. J. Weinberger, "Cheap Dynamic Instruction Counting," *Bell Systems Technical Journal*, vol. 63, pp. 1815-1826, October 1984.

[12] AT&T, *UNIX System V/386 Programmer's Reference Manual*. Englewood Cliffs, NJ: Prentice Hall, 1988.

[13] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Englewood Cliffs, New Jersey: Prentice-Hall, 1978.

[14] K. A. El-Ayat and R. K. Agarwal, "The Intel 80386 - Architecture and Implementation," *IEEE Micro*, pp. 4-22, December 1985.

[15] C. B. Stunkel, "Linear Optimization Via Message-Based Parallel Processing," *Proc. Int. Conf. on Parallel Processing*, vol. III, pp. 264-271, August 1988.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

# REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

| | |
|---|---|
| 1a. REPORT SECURITY CLASSIFICATION<br>Unclassified | 1b. RESTRICTIVE MARKINGS<br>None |
| 2a. SECURITY CLASSIFICATION AUTHORITY | 3. DISTRIBUTION / AVAILABILITY OF REPORT<br>Approved for public release; distribution unlimited |
| 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S)<br>CSG 94 UILU-ENG-88-2261 | 5. MONITORING ORGANIZATION REPORT NUMBER(S) |
| 6a. NAME OF PERFORMING ORGANIZATION<br>Coordinated Science Lab<br>University of Illinois<br>6b. OFFICE SYMBOL (If applicable)<br>N/A | 7a. NAME OF MONITORING ORGANIZATION<br>NASA |
| 6c. ADDRESS (City, State, and ZIP Code)<br>1101 W. Springfield Ave.<br>Urbana, IL 61801 | 7b. ADDRESS (City, State, and ZIP Code)<br>NASA Langley Research Center<br>Hampton, VA 23665 |
| 8a. NAME OF FUNDING / SPONSORING ORGANIZATION<br>NASA<br>8b. OFFICE SYMBOL (If applicable) | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER<br>NASA NAG 1-613 |
| 8c. ADDRESS (City, State, and ZIP Code)<br>See 7b. | 10. SOURCE OF FUNDING NUMBERS<br>PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO. |

11. TITLE (Include Security Classification)
TRAPEDS: Producing Traces for Multicomputers Via Execution-Driven Simulation

12. PERSONAL AUTHOR(S) Craig B. Stunkel and W. Kent Fuchs

| 13a. TYPE OF REPORT | 13b. TIME COVERED | 14. DATE OF REPORT (Year, Month, Day) | 15. PAGE COUNT |
|---|---|---|---|
| Technical | FROM _____ TO _____ | September 1988 | 20 |

16. SUPPLEMENTARY NOTATION

| 17. COSATI CODES | | | 18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) |
|---|---|---|---|
| FIELD | GROUP | SUB-GROUP | Performance evaluation, address trace, trace-based simulation, execution driven simulation, multiprocessors |
| | | | |
| | | | |

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Trace-driven simulation is an important aid in performance analysis of computer systems. Capturing address traces for these simulations is a difficult problem for single processors and particularly for multicomputers. Even when existing trace methods can be used on multicomputers, the amount of collected data typically grows with the number of processors, so I/O and trace storage costs increase. A new technique is presented in this paper which modifies the executable code to dynamically collect the address trace from the user code and analyzes this trace during the execution of the program. This method helps resolve the I/O and storage problems and facilitates parallel analysis of the address trace. If a trace stored on disk is desired, the generated trace information can also be written to files during execution, with a resultant drop in program execution speed. An initial implementation on the Intel iPSC/2 hypercube multicomputer is detailed, and sample simulation results are presented. The effect of this trace collection method on execution time is illustrated.

| 20. DISTRIBUTION / AVAILABILITY OF ABSTRACT | 21. ABSTRACT SECURITY CLASSIFICATION |
|---|---|
| ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS | Unclassified |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL | 22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL |

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE



UNCLASSIFIED