AFRL-RI-RS-TR-2008-139
Final Technical Report
May 2008

EVALUATION OF THE IMPLICATIONS OF NANOSCALE ARCHITECTURES ON CONTEXTUAL KNOWLEDGE DISCOVERY AND MEMORY: SELF-ASSEMBLED ARCHITECTURES AND MEMORY

Duke University

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

FOR THE DIRECTOR:

/s/	/s/
THOMAS E. RENZ Work Unit Manager	JAMES A. COLLINS, Deputy Chief Advanced Computing Division Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* MAY 08	**2. REPORT TYPE** Final	**3. DATES COVERED** *(From - To)* Dec 04 – Dec 07

4. TITLE AND SUBTITLE EVALUATION OF THE IMPLICATIONS OF NANOSCALE ARCHITECTURES ON CONTEXTUAL KNOWLEDGE DISCOVERY AND MEMORY: SELF-ASSEMBLED ARCHITECTURES AND MEMORY	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER FA8750-05-2-0018
	5c. PROGRAM ELEMENT NUMBER 62702F
6. AUTHOR(S) Chris Dwyer	**5d. PROJECT NUMBER** 459T
	5e. TASK NUMBER 20
	5f. WORK UNIT NUMBER 02
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Duke University 334 N. Building Research Dr. Durham NC 27708-9900	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/RITC 525 Brooks Rd Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-RI-RS-TR-2008-139

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# WPAFB 08-3187

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Computing systems with advanced situational awareness and the ability to use contextual knowledge to interpret sensor data have the potential to be instrumental in many contexts. This project developed three systems to query a database with immense numbers of objects and rich sets of contextual relationships. In particular, large-scale content addressable memory systems provide a better solution to the knowledge discovery problem than conventional general-purpose memory systems. This project studied three systems: 1) a conventional system, 2) a conventional system optimized for online (i.e. real-time) use, and 3) a novel DNA self-assembled nanoelectronic system. The project developed tools for DNA self-assembly to provide simulation capabilities for evaluating the three systems and the data has shown that significant performance enhancements can be achieved by optimization. Further, when self-assembling technologies mature they will be able to achieve greater performance due to the massive parallelism inherent in the knowledge discovery problem.

15. SUBJECT TERMS
Associative Memory, Content Addressable Memory, Nanoelectronics, DNA Self-Assembly

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Thomas Renz
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	82	**19b. TELEPHONE NUMBER (*Include area code*)** N/A

Table of Contents

List of Figures

List of Tables

1. Summary

Computing systems with advanced situational awareness and the ability to use contextual knowledge to interpret sensor data have the potential to be instrumental in many contexts. In particular, large-scale content addressable memory systems provide a better solution to the knowledge discovery problem than conventional general-purpose memory systems. This project developed three systems to query a database with immense numbers of objects and rich sets of contextual relationships; (i) a conventional system, (ii) a conventional system optimized for online (i.e. real-time) use, and (iii) a novel Deoxyribonucleic Acid, DNA self-assembled nanoelectronic system. The project developed tools for DNA self-assembly to provide simulation capabilities for evaluating the three systems and the data has shown that significant performance enhancements can be achieved by optimization of the memory access pattern. Further, when self-assembling technologies mature they will be able to achieve greater performance due to the massive parallelism inherent in the knowledge discovery problem.

The goal of this project was to evenly re-evaluate the methods used by experts in the knowledge discovery and context aware application field with respect to emerging nanoscale fabrication capabilities. The performance predicted by industry experts for conventional silicon devices (e.g., Complimentary Metal Oxide Semiconductor, CMOS) and the anticipated performance we have simulated for DNA self-assembled nanoelectronic devices will help put each system on a level footing with respect to each other.

The major challenge in knowledge discovery and context aware applications is to support the querying of a database with immense numbers of objects and rich sets of contextual relationships. The optimal form of this process is NP-hard because new information must be considered against all previously encountered information. As the database grows, more and more cross-references will be generated and this can potentially (i.e., in the worst case) take exponential time in the number of objects contained by database.

The emergence of nanoscale architectures and fabrication methods has changed the design space for these applications by enabling novel forms of "at-fabrication" computation and massively parallel architectures. These developments make a case for the re-evaluation of current best practices in context aware memory system design. In particular, large-scale content addressable memory systems, enabled by nanoscale self-assembly, may provide a better solution to the knowledge discovery problem than conventional general purpose memory systems that use centralized indexing schemes.

This project evaluated the three memory systems within the design space of context aware and knowledge discovery architectures. This evaluation used target CMOS technology from the International Technology Roadmap for Semiconductors prediction for 2012 [1, 2], and future DNA self-assembled technology as a competitive alternative [3-6]. A variety of metrics were explored from the literature to determine a significant figure of merit for each system.

The first system is a conventional CMOS-based general-purpose architecture that assigns context during post-query processing. The second system we studied stores contextual information soon after the sensor or input data is stored. That is, the searching required to create the contextual information occurs when new information is stored in the memory. This method has the advantage that the query time can be much faster than a conventional system but the time to store new information will take just as long as the conventional system's query time (since context assignment occurs during storage.) Therefore, the rate-limiting step for this system is still the context assignment. This system is most like a conventional data-mining system where additional fields (i.e., tags or hints) are included to facilitate context assignment.

The third system we studied was a novel system enabled by DNA self-assembly. This system can use self-assembled systems to perform a distributed context assignment during information storage but incur a much smaller penalty than the pre-retrieval system because of the massive parallelism inherent in molecular-scale self-assembly. The DNA self-assembled system leverages the short query time of the pre-retrieval system against the massive parallelism of self-assembly to reduce storage times. The result is a balanced system with equal storage and retrieval times. However, we have found that a significant gap exists between what can be built in our laboratory and what is required by a contextual knowledge memory. Thus, we have shown through simulation significant performance improvements over a conventional system but significant progress in the technology remains untapped.

2. Methods, Assumptions, and Procedures

DNA is an attractive substrate to investigate for applications in molecular-scale computing. The precise binding rules of DNA enable the creation of nanostructures with minimum pitch on the order of a few nanometers. Further, these nanostructures can be used to place and interconnect nanoscale components with molecular-scale precision. Thus, DNA self-assembly is an enabling technology for new computing paradigms [1-19].

Double stranded DNA structure is stable when the base pairs are “complementary”, i.e., if A pairs with T and G pairs with C. The central theme in the use of self-assembly for nanoscale fabrication is the application of external control over an otherwise spontaneous reaction to direct its outcome [20]. This control directs the assembly of materials into structures that are interesting and relevant to a target design problem. In the context of computer system fabrication, self-assembly is used to direct the formation of switching devices and wires to create logic circuitry, memory, and I/O interfaces. We can control the reaction by designing synthetic DNA strands to interact at specific temperatures (called melting temperatures) by careful choice of their nucleotide sequences. Specification of the strand sequences provides control over the self-assembly process by establishing the melting temperature of the strands which determines the formation of structures (through complementarity). Sequence design is important because it determines many aspects of the target DNA nanostructure (e.g., geometry and stability).

Complex designs are often created using a relatively small set of common building blocks—called motifs. DNA self-assembly can exploit this same design principle to hierarchically create more sophisticated aperiodic structures. For DNA there are many possible motifs, however we focus on only a few in the context of our resonance energy transfer logic. Motifs include junctions that enable three or more double stranded helices of DNA to interact and thus form specific structures (e.g., a triangle, a corner, etc.) Another important motif is a single strand of DNA protruding from a double stranded helix—called a sticky-end.

Two motifs with complementary sequences on their sticky-ends will bind to form a composite motif. These composite motifs may also have embedded sticky-end motifs and thus can also bind with other composite motifs to form another, larger, composite motif. This results in a hierarchical structure for motifs. Hierarchical DNA self-assembly provides the fabrication characteristics (low-cost, nm resolution) necessary for molecular-scale computation. However, the substrate must be complemented by suitable molecular-scale devices.

The following section (and its sub-sections) describe in detail the final conclusions and findings of this project.

3. Major Results and Discussion

3.1 Data Mining and Contextual Knowledge Discovery

Data mining is the process of extracting or mining interesting knowledge from large amounts of data stored in databases or other information stores. A fundamental and essential problem in data mining is to discover frequent patterns or itemsets: given a data base of item transactions, find all itemsets that occur in at least a user-specified percentage of the total transactions in the database (i.e., has a specified support level). Frequent itemset mining is useful for many data mining capabilities such as discovery of association rules [1], strong rules, correlations, sequential rules [2], episodes [12], multi-dimensional patterns, partial periodicity [9], and many other important discovery tasks. These data mining capabilities are useful for practical applications such as market basket analysis, inferring patterns from web access logs, and network intrusion detection among others.

Frequent itemset mining generates a very large number of patterns which may reduce not only the efficiency but also effectiveness of mining, since it generates numerous redundant patterns. Furthermore, users must sift through a large number of mined patterns to find useful ones. Closed frequent itemset (CFI) mining [14], mines only those frequent itemsets having no proper superset with the same support, which can lead to orders of magnitude smaller result set than mining frequent itemsets [15]. CFI mining retains all the information of frequent itemset mining, as it is straightforward to generate all the frequent itemsets from the closed frequent itemsets.

In this report we explore the memory hierarchy performance of a state-of-the-art CFI algorithm—called FPclose[1] [8]—using real world datasets. We use a combination of tools (OProfile [11], VTUNE [16], Simplescalar and iostat) to gain insight into FPclose's memory hierarchy performance for both sparse and dense real world datasets on both Pentium III and Pentium 4 systems. The characterization results indicate that cache behavior becomes a critical performance bottleneck as the support level decreases. Simulation experiments show that gains in the Instructions per cycle, IPC are seen for a system with perfect memory hierarchy. Increase in width of the instruction window has almost no effect on the IPC.

One specific function CFI_tree::insert(bool*,…) accounts for up to 69% of the L1 data cache misses, 86% of the L2 data cache misses, and 69% of the execution time for moderate to low support levels. This function is unique to the closed frequent itemset problem. Symbolic profiling and source code inspection reveal that CFI_tree::insert(bool*,…) function incurs most of its cache misses while traversing the nodes of a prefix tree (or closed frequent pattern tree) at a specific level (right sibling pointer dereferences). This motivates the application of well-known data structure modifications to improve cache performance: 1) padding & aligning, 2) node clustering [4]. Our characterization results also reveal that the average number of nodes accessed during the sibling traversal is data dependent. This motivates a final optimization that dynamically increases node clustering as more nodes are allocated in a specific level of the prefix tree.

We evaluate the data structure modifications by measuring execution cycles on Pentium III and Pentium 4 systems and comparing with execution cycles of base case. Our results show that padding and aligning have no significant impact on performance. Node clustering achieves performance gains of up to 79% on Pentium III systems and 65% on Pentium 4. Finally, we show that the dynamic allocation technique provides speedups up to 81% on Pentium III system and 67% on Pentium 4. The maximum difference in speedup of the best static sized node clustering scheme at each data point and the dynamic scheme was 6% on Pentium III and 8% on Pentium 4. This difference was in dense datasets, for sparse datasets dynamic scheme performed as well as or better than the best static sized node clustering scheme at each data point in many cases for both systems. For dense datasets node clustering produces the greatest improvements in performance on both Pentium systems.

3.1.1 Closed Frequent Item Set Mining

Let I = (i_1, i_2 ..., i_n) be a set of items. An itemset is a non-empty subset of I. A transaction is represented by a tuple of the form (tid, X), where *tid* is the transaction identifier and *X* is an itemset. A transaction database (*TDB*) is a set of transactions. An itemset *X* is contained in transaction (tid, Y) if X is a subset of Y. Given a transaction database *TDB*, the support of an itemset *X* is the number of transactions in *TDB* which contains *X*. An itemset *X* is called a frequent itemset if its support is no less than the minimum support threshold. Frequent itemset mining consists of finding all the frequent itemsets. An itemset *X* is a closed itemset if there exists no itemset *X'* such that *X'* is a proper superset of *X*, and every transaction containing *X* also contains *X'*. A closed itemset *X* is frequent if its support is greater than or equal to minimum support threshold. Closed frequent itemset mining consists of finding all closed frequent itemsets.

We chose the state-of-the-art FPclose [8] algorithm as our closed frequent itemset implementation. FPclose won the FIMI'03 best implementation award [17]. FPclose uses variation of the FP-tree (Frequent Pattern tree, also called a prefix tree) structure for checking the closedness of frequent itemsets. The FP-tree itself has been shown to be one of the most efficient data structures for mining frequent patterns. A novel technique which employs an array, greatly improves the performance of the algorithm while operating on the FP-tree.

3.1.2 Performance Characterization

This section describes the tools used, the system configuration, the datasets, the characterization procedure and its results.

3.1.2.1 Datasets

For this study we use five different datasets that can be categorized into sparse, moderate, and dense. Dense datasets have large number of long transactions. Sparse datasets have predominantly short transactions. Long transaction means that the average number of items per transaction is high, vice versa for short transactions. We consider five real datasets; retail [3] and bmspos [18] are sparse, pumsb and accidents [6] are dense, while we consider kosarak moderate. Description about the datasets is provided in Table 1. Characteristics of the dataset are shown in Table 2.

Table 1: Dataset Description

Retail	Retail market basket data set supplied by a anonymous Belgian retail supermarket store
Bmspos	The BMS-POS dataset contains several years' worth of point-of sale data from a large electronics retailer.
Kosarak	Click-stream data of a Hungarian on-line news portal
pumsb	Census data (from IBM Almaden Research Center)
Accidents	Traffic accident information for the region of Flanders (Belgium) for the period 1991-2000

Table 2: Dataset Characteristics

Dataset	No. of Items	Avg. Transaction Length	No. of Transactions
Retail	16,469	10.3	88,162
Bmspos	1,657	6.5	515,597
Kosarak	41,270	8.1	990,002
Pumsb	2,113	74.0	49,046
Accidents	468	33.8	340,183

3.1.2.2 Tools

We profiled the FPclose algorithm using VTune and OProfile. VTune and OProfile are system-wide profilers, capable of profiling all running code at low overhead. They consist of a kernel driver and a daemon for collecting sample data, and several post-profiling tools. The hardware performance counters of the CPU are leveraged to enable profiling of a wide variety of interesting statistics. All code is profiled: hardware and software interrupt handlers, kernel modules, the kernel, shared libraries, and applications. We chose OProfile version 0.8.2 and VTune version 7.2. VTune provides more events for Pentium 4 as opposed to Oprofile so we chose VTune for profiling workloads on Pentium 4.

SimpleScalar tool set version 3 was used for analyzing the performance of the memory hierarchy. The SimpleScalar tool set is a system software infrastructure used to build modeling applications for program performance analysis, detailed micro-architectural modeling, and hardware-software co-verification. Modeling applications can simulate real programs running on a range of modern processors and systems.

We used time command and iostat package to study the I/O characteristics of different workloads.

3.1.3 System Configuration

We characterize the performance of FPclose executing on a Pentium III system running Debian Linux with 2.6.10 kernel and a Pentium 4 system running Red hat Linux with kernel 2.4.21. The hardware configuration of these systems is outlined in Tables 3 and 4 respectively.

Table 3: Pentium III System configuration

Processor	Intel Pentium III CPU family, 1400 MHz
L1 data cache	16 KB, 4-way set associative, 32 byte line size
L1 instruction cache	16 KB, 4-way set associative, 32 byte line size
L2 unified cache	512 KB, 8-way set associative, 32 byte line size
Instruction TLB	4 KB pages, 4-way set associative, 32 entries
Data TLB	4 KB pages, 4-way set associative, 64 entries
Memory	1 GB

Table 4: Pentium 4 System configuration

Processor	Intel Pentium 4 CPU family, 3 GHz
L1 data cache	8 KB, 4-way set associative, 64 byte line size
Trace Cache	12K-micro-op, 4-way set-assoc
L2 unified cache	1024 KB, 8-way set associative, 64 byte line size
Instruction TLB	4 KB pages, 64 entries
Data TLB	4 KB pages, 64 entries
Memory	1 GB

3.1.4 Methodology

FPclose is executed on all the datasets with different minimum support thresholds as input. Minimum support thresholds ranged from levels where only a handful of itemsets were generated, to as low a level as possible. For accident, pumsb, and kosarak dataset, the lowest support threshold was bounded by the level at which the process would abort due to lack of physical memory. For each of the workloads we measure the performance for computing all the closed frequent itemsets (CFIs), not for outputting them. We ensure the system is lightly loaded during profiling. The FPclose executable and the dataset were stored on the local disk to reduce the impact of NFS (Network File Service) traffic on the results. The OProfile or VTune daemon, configured to measure different events, runs on the same system to do profiling of the workloads.

For Oprofile (used to profile on the Pentium III), we measure IPC (Instructions per Cycle) using the events CPU_CLK_UNHALTED and INST_RETIRED, L1 data cache miss rates using the events DCU_LINES_IN and DATA_MEM_REFS, and global L2 cache miss rates using events L2_LINES_IN and DATA_MEM_REFS.

For VTune (used to profile on the Pentium 4), we measure IPC using the events Clockticks and Instructions Retired, L1 load data cache miss rates using the events 1st-Level Cache Load Misses Retired and Loads Retired, and global L2 cache load miss rates using events 2nd-level Cache Load Misses Retired and Loads Retired, Data Translation Look aside buffer (DTLB) load miss rate using events DTLB Load Misses and Loads Retired. VTune does a configuration run before calculating actual statistics. We measured only the load miss rates on the Pentium 4 due to limited event counters. Although care must be taken in interpreting the Pentium 4 miss rates, our results show that the trends are consistent with those from the Pentium III. We perform both symbol level and process level profiling for each of the workloads.

Symbol level profiling enables mapping performance metrics to program source, specifically to each of the functions of the FPclose program. To achieve accurate source mapping, symbolic profiling uses a slightly different executable which was compiled without optimization to preserve accurate symbol table information.

Different workloads were executed under time command to study their I/O characteristics. Time command measures CPU utilization. As the system is lightly loaded, a drop in CPU utilization can be attributed to the fact that the CPU is idle as the process is waiting for disk I/O. Page fault statistics provide further validation. iostat package helps us in studying the time behavior of the I/O characteristics of a workload.

In order to measure the impact of memory hierarchy on the overall performance, we simulated some of the workloads using SimpleScalar. SimpleScalar can simulate alpha or PISA (Portable Instruction Set Architecture) binaries. Using gcc cross compiler and assembler, the PISA binary for the FPclose algorithm was built. A system configuration that would resemble as closely as possible the Pentium III system on which we had profiled our workloads was chosen. The system configuration chosen is given in Table 5.

Table 5: SimpleScalar System Description

Instruction fetch queue size	32
Extra branch misprediction latency	12
Branch Predictor	bimodal predictor (Branch Target Buffer (BTB) w/ 2 bit counters)
Direct mapped BTB size	4096
BTB configuration	512 sets, 4-way set associative
Return address stack size	32
Decoder bandwidth	4 insts/cycle
Issue bandwidth	4 insts/cycle
Permit instruction issue after mis-speculation	False
Instruction commit bandwidth	4 insts/cycle
Register Update Unit(RUU) size	64
Load/store queue size	32
L1 data cache configuration	16 KB, 4-way set associative, 32 byte line size, hit latency 3 cycles
L1 instruction cache	16 KB, 4-way set associative, 32 byte line size, hit latency 3 cycles
L2 unified cache	512 KB, 8-way set associative, 32 byte line size, hit latency 25 cycles
Instruction TLB	4 KB pages, 4-way set associative, 32 entries, miss latency 30 cycles
Data TLB	4 KB pages, 4-way set associative, 64 entries, miss latency 30 cycles
Memory access latency (first, rest)	150, 30 cycles

Width of memory bus	16 bytes
Number of integer ALUs	4
Number of integer multiplier/dividers	1
Number of first-level cache ports	2
Number of Floating point ALUs	4
Number of Floating point multiplier/dividers	1

For the retail and kosarak datasets we did detailed simulation of the whole run for a particular support level. For the accident dataset we fast-forwarded a certain number of instructions and then did detailed simulation after that for the next 50 trillion instructions. Only functional simulation is done during fast-forwarding. This helps in reducing the simulation time. On the basis of symbol profiling we found a bottleneck function. We fast-forwarded up to 100 million instructions before the point at which that bottleneck function is invoked for the first time. We then started doing detailed simulation in order to warm up the caches before we invoke that function for the first time. We followed a similar procedure for pumsb dataset.

Apart from gathering statistics for the configuration given in Table 5, we also gathered statistics for a system having perfect memory hierarchy. We also try to see the effect of a wide instruction window on the performance of FPclose algorithm by varying the size of the register update unit in SimpleScalar. We did this simulation for the accident dataset.

3.1.5 Profiling Results

In this section we present the profiling results for the different workloads. For each dataset we perform process level profiling to obtain an overall performance characterization on both Pentium systems and symbolic profiling to obtain insight into the cause of performance bottlenecks.

3.1.6 Performance Characterization

In this section we present process level performance characterization results for all five datasets. Figure 1 shows the number of closed frequent itemsets (CFIs) present in the retail dataset for various support thresholds. Figure 2 shows the cycles required for finding all the CFIs at various support thresholds. Figure 3 shows the IPC for the retail dataset at different support thresholds. Figure 4 shows the L1 data cache and L2 unified cache global miss rates for the retail dataset at various minimum support thresholds. Figures 5-7 show the corresponding measurements for Pentium 4 system. Figure 8 shows the DTLB load miss rates for retail dataset on Pentium 4. We report only the load cache and TLB miss rates for the Pentium 4. The TLB miss rate cannot be obtained on the Pentium III.

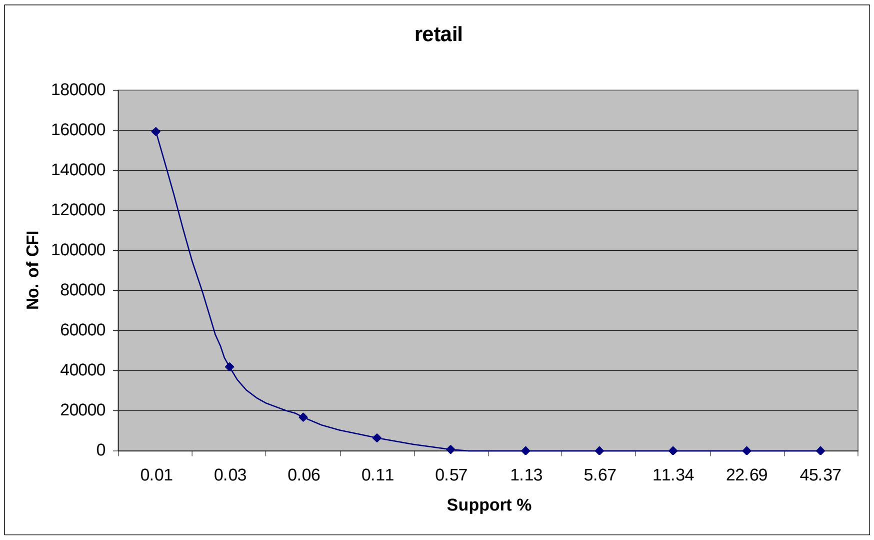


Figure 1: Distribution of Closed Frequent Itemsets

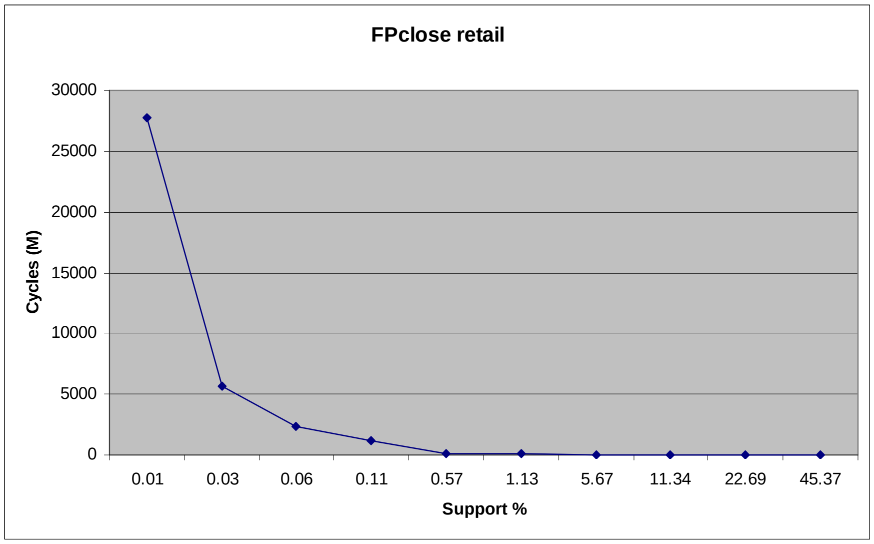


Figure 2: Execution cycles (Pentium III)

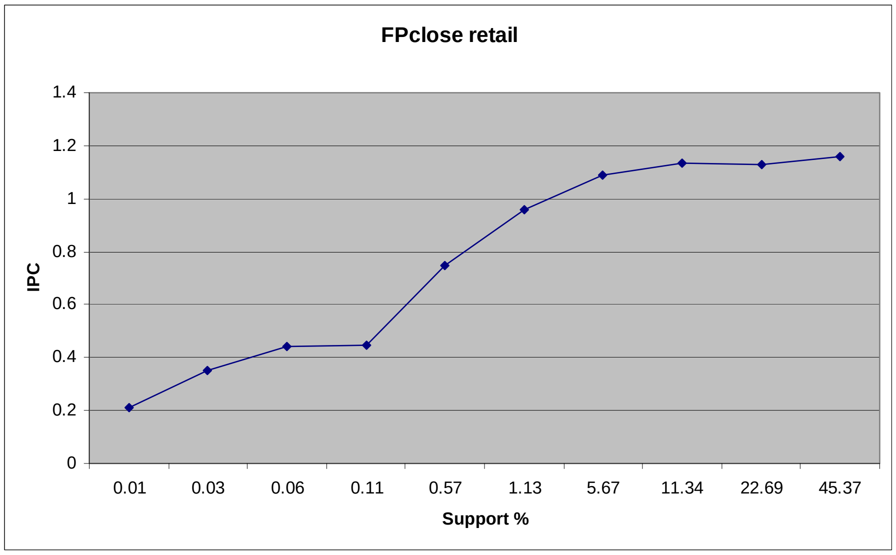


Figure 3: Instruction per Cycle (IPC) (Pentium III)

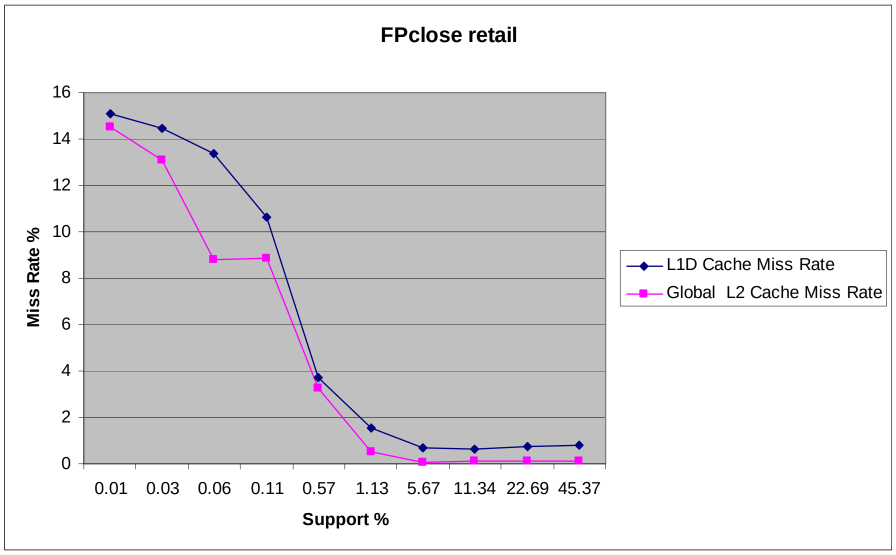


Figure 4: L1 Data and L2 Cache Miss Rates (Pentium III)

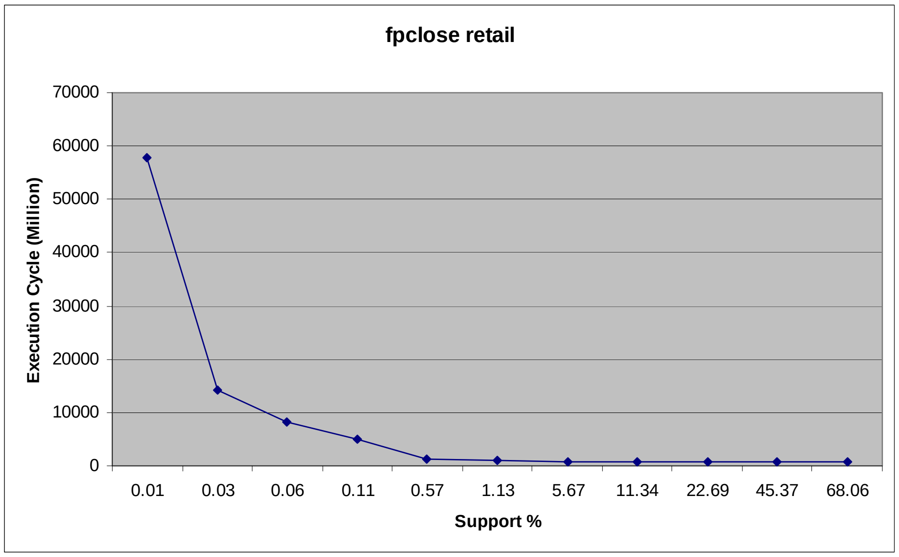


Figure 5: Execution cycles (Pentium 4)

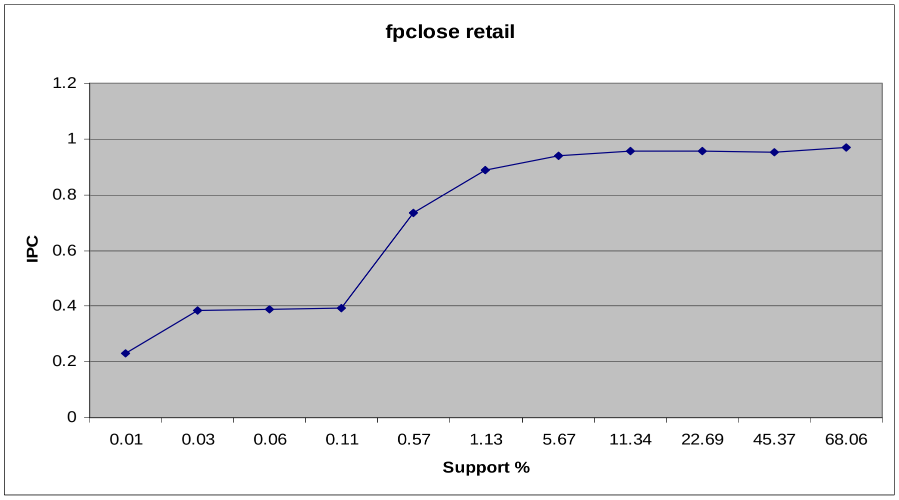


Figure 6: Instruction per Cycle (IPC) (Pentium 4)

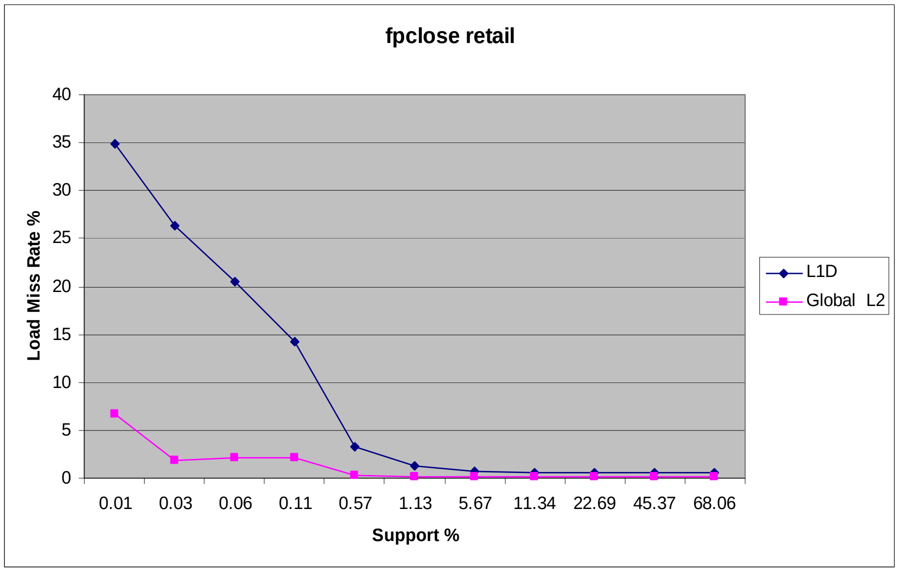


Figure 7: L1 Data and L2 Cache Load Miss Rates (Pentium 4)

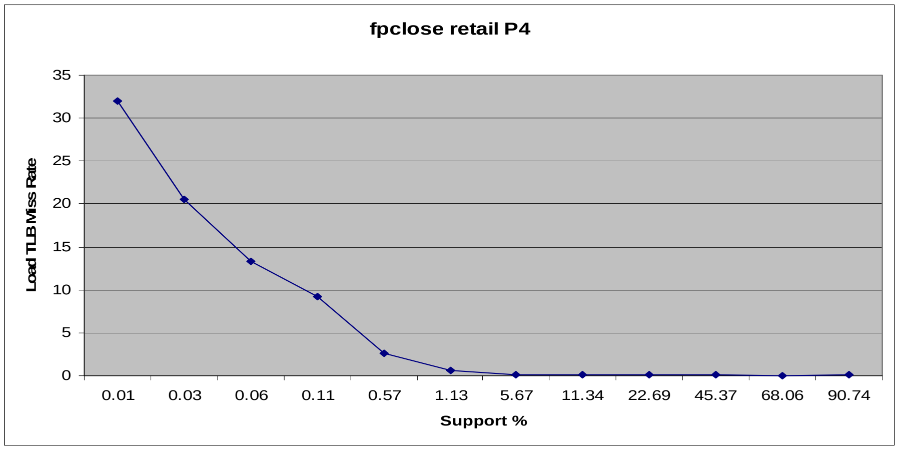


Figure 8: Data TLB Load Miss Rates (Pentium 4)

Figure 9 shows the number of closed frequent itemsets (CFIs) present in the bmspos dataset for various support thresholds. Figure 10 shows the cycles required for finding all the CFIs. Figure 11 shows the IPC for the bmspos dataset at different support thresholds. Figure 12 shows the corresponding L1 data cache and L2 unified cache global miss rates for the bmspos dataset. Figures 13-15 show the corresponding measurements for Pentium 4 system. Figure 16 shows the DTLB load miss rates for bmspos dataset on Pentium 4.

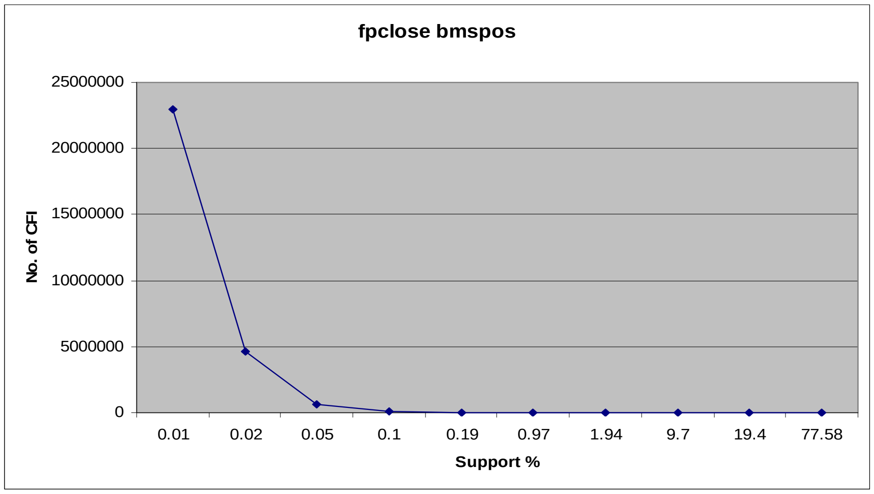


Figure 9: Distribution of Closed Frequent Itemsets

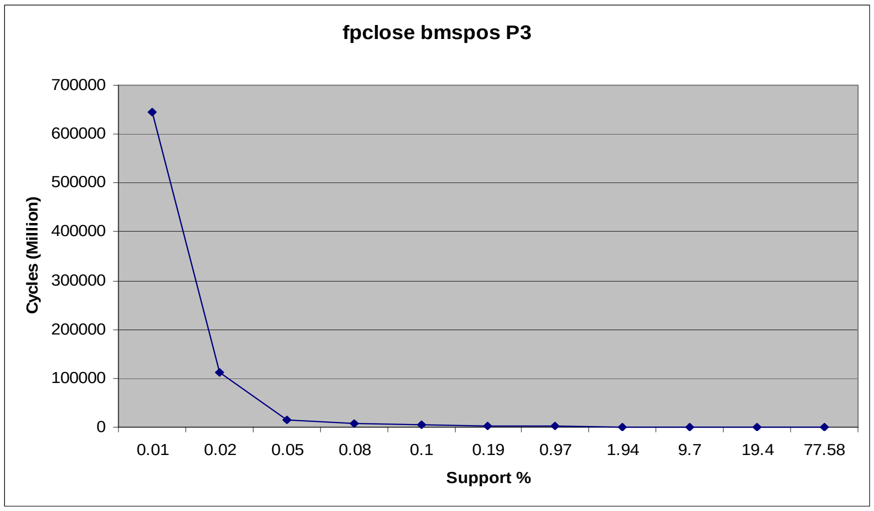


Figure 10: Execution cycles (Pentium III)

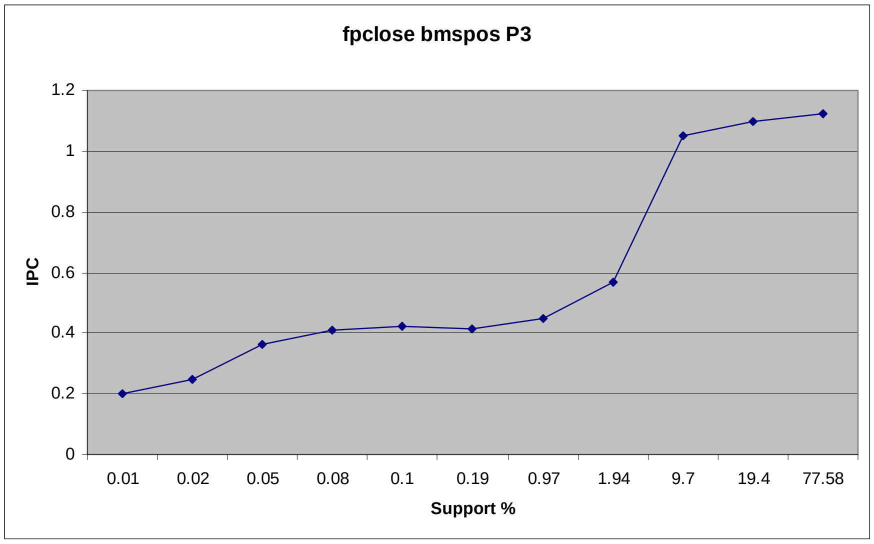


Figure 11: Instruction per Cycle (IPC) (Pentium III)

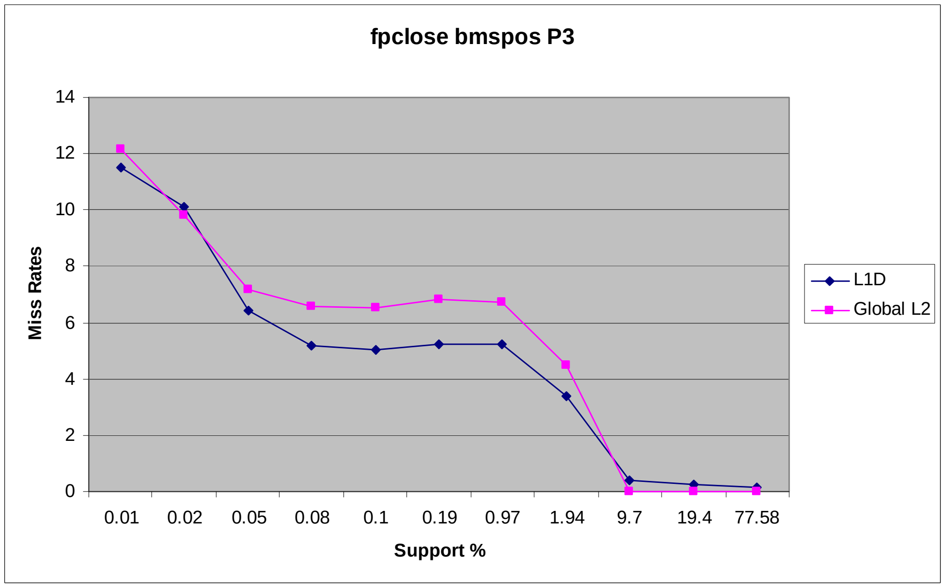


Figure 12: L1 Data and L2 Cache Miss Rates (Pentium III)

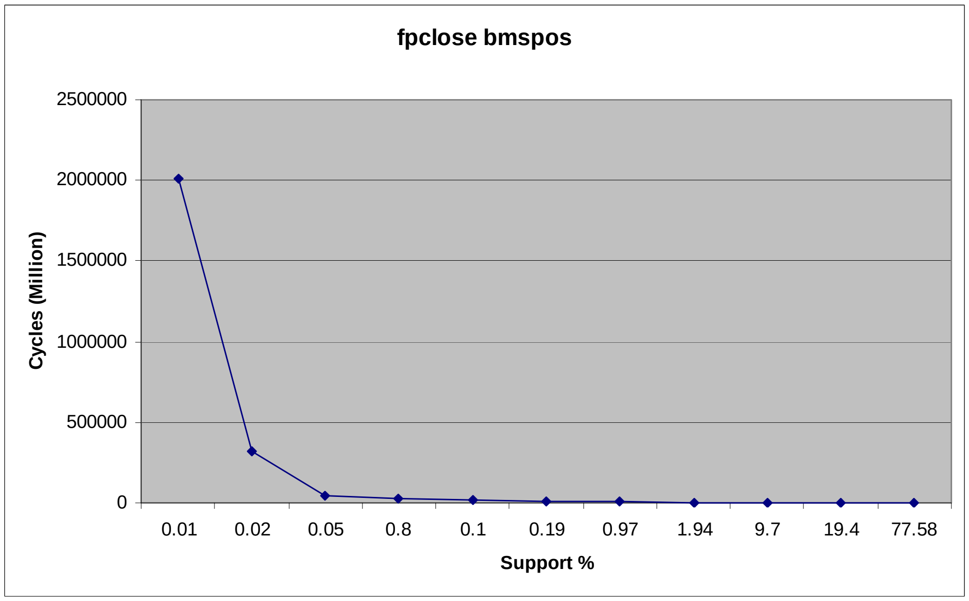


Figure 13: Execution cycles (Pentium 4)

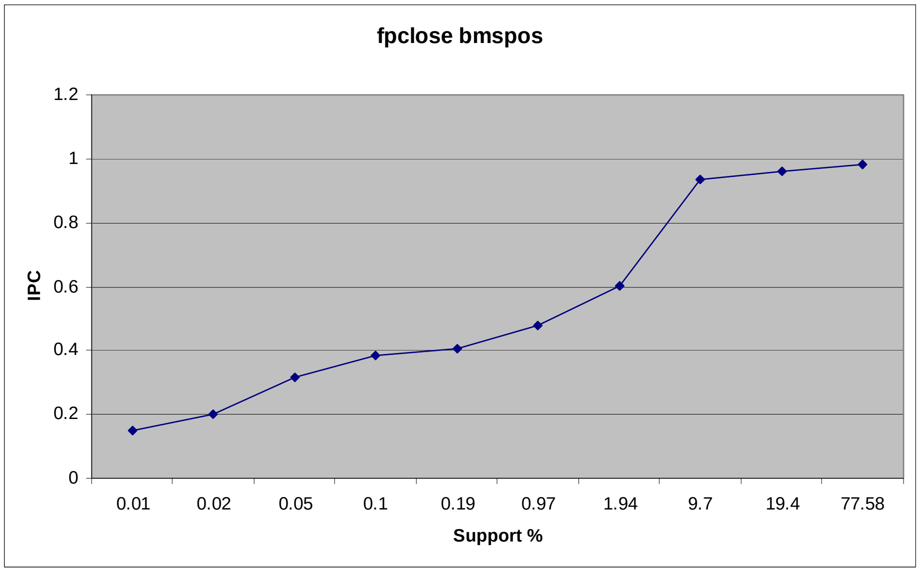


Figure 14: Instruction per Cycle (IPC) (Pentium 4)

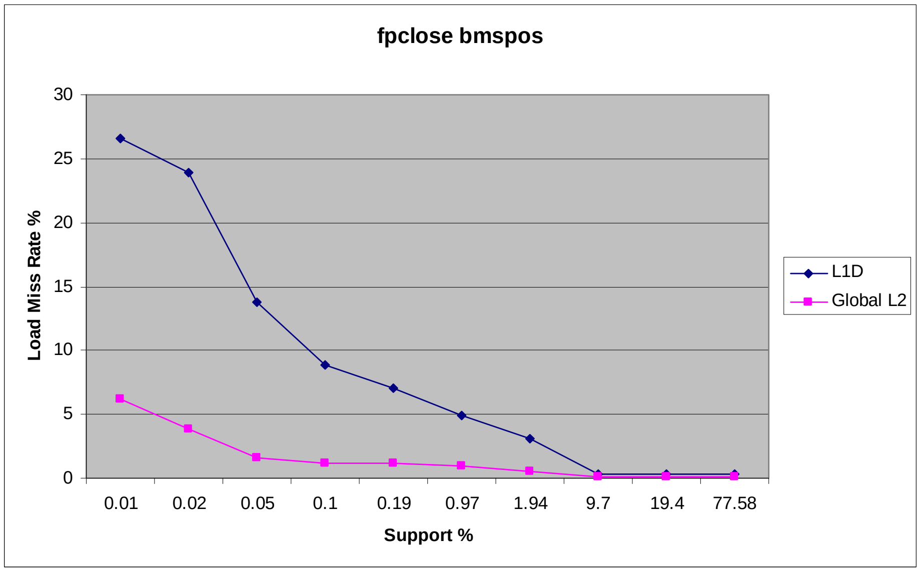


Figure 15: L1 Data and L2 Cache Load Miss Rates (Pentium 4)

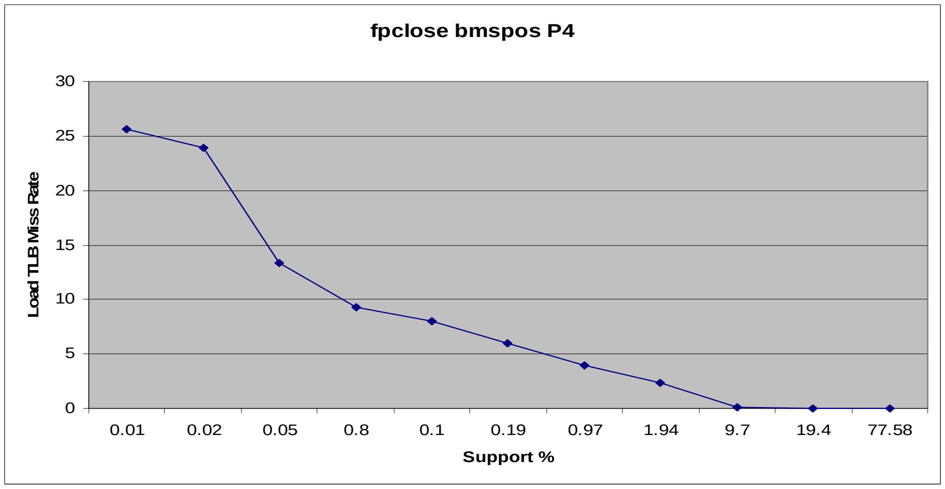


Figure 16: Data TLB Load Miss Rates (Pentium 4)

Figure 17 shows the number of closed frequent itemsets (CFIs) present in the accident dataset for various support thresholds. Figure 18 shows the cycles required for finding all the CFIs. Figure 19 shows the IPC for the accident dataset at different support thresholds. Figure 20 shows the corresponding L1 data cache and L2 unified cache global miss rates for the accident dataset. Figures 21-23 show the corresponding measurements for Pentium 4 system. Figure 24 shows the DTLB load miss rates for accident dataset on Pentium 4.

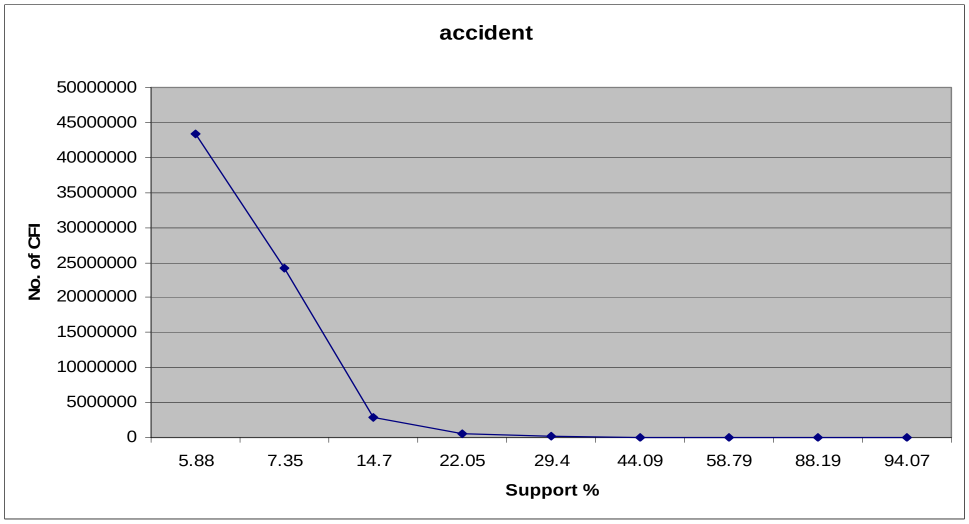


Figure 17: Distribution of Closed Frequent Itemsets

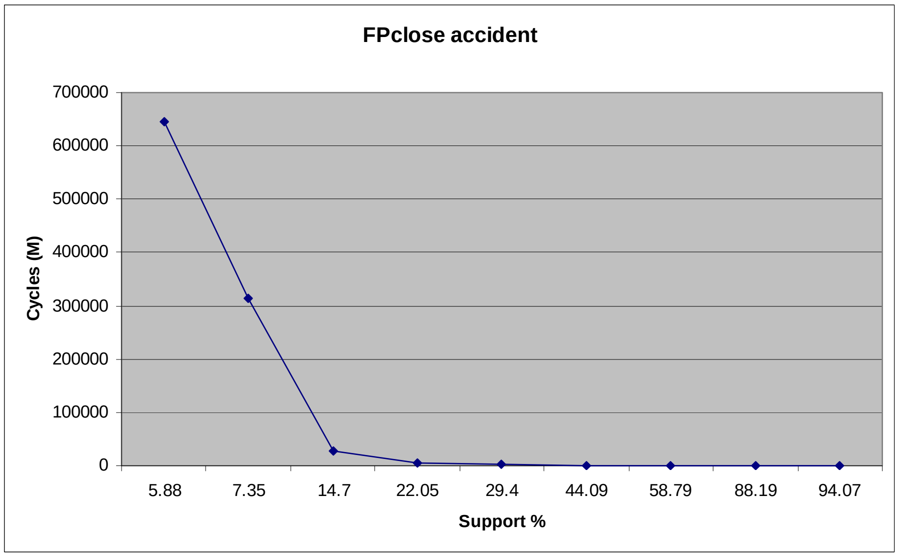


Figure 18: Execution cycles (Pentium III)

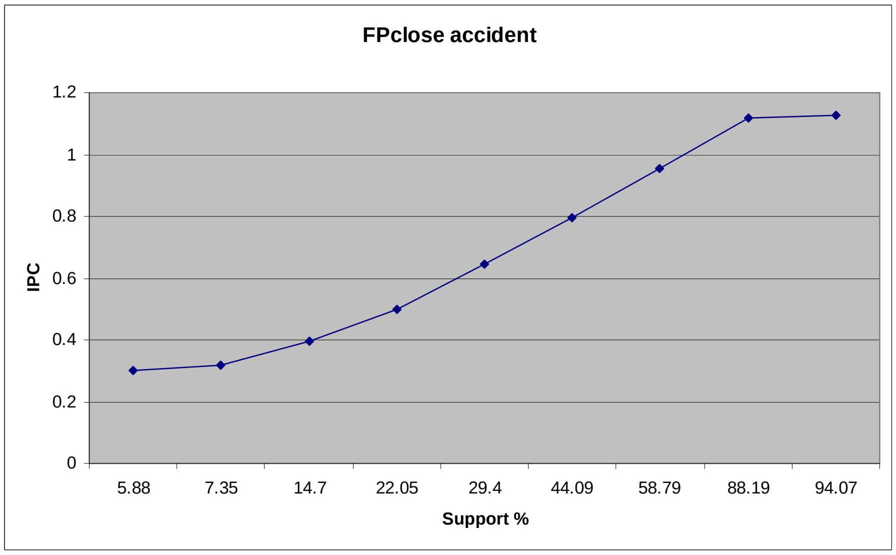


Figure 19: Instruction per Cycle (IPC) (Pentium III)

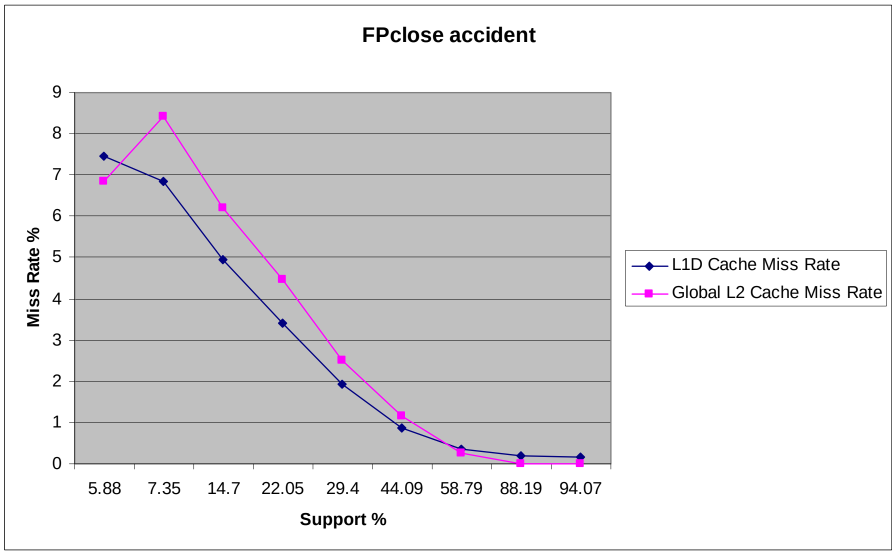


Figure 20: L1 Data and L2 Cache Miss Rates (Pentium III)

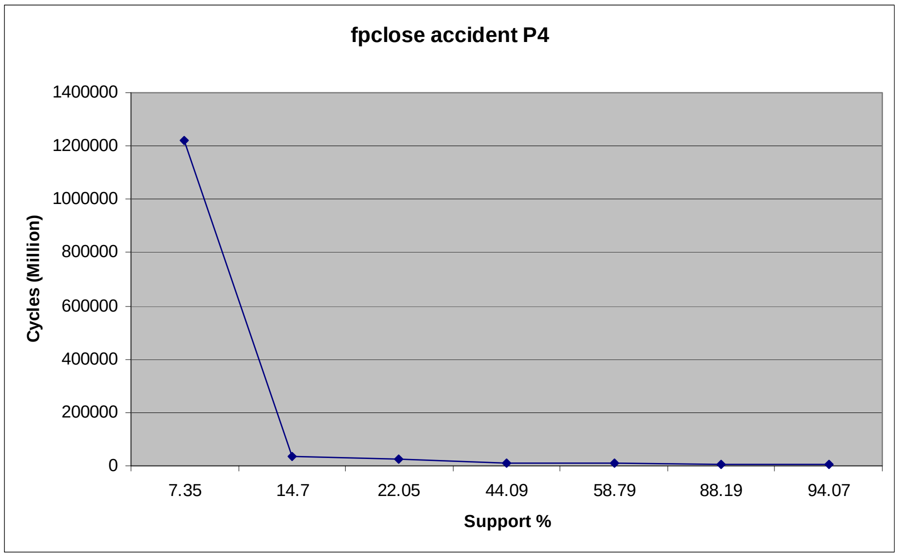


Figure 21: Execution cycles (Pentium 4)

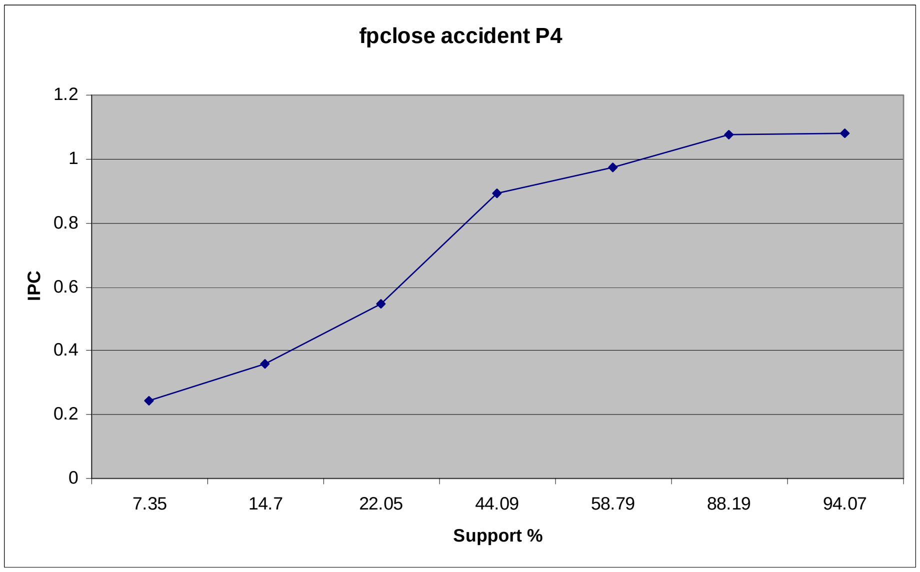


Figure 22: Instruction per Cycle (IPC) (Pentium 4)

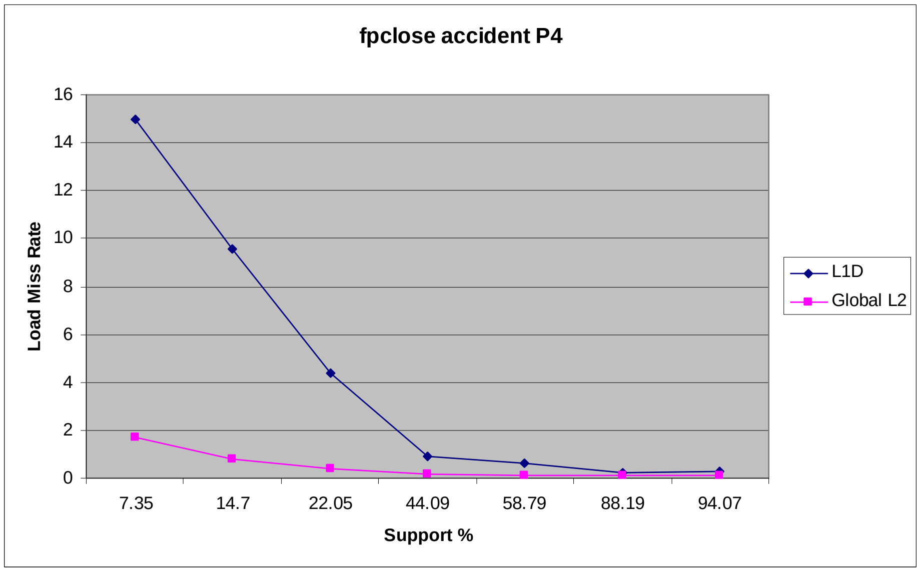


Figure 23: L1 Data and L2 Cache Load Miss Rates (Pentium 4)

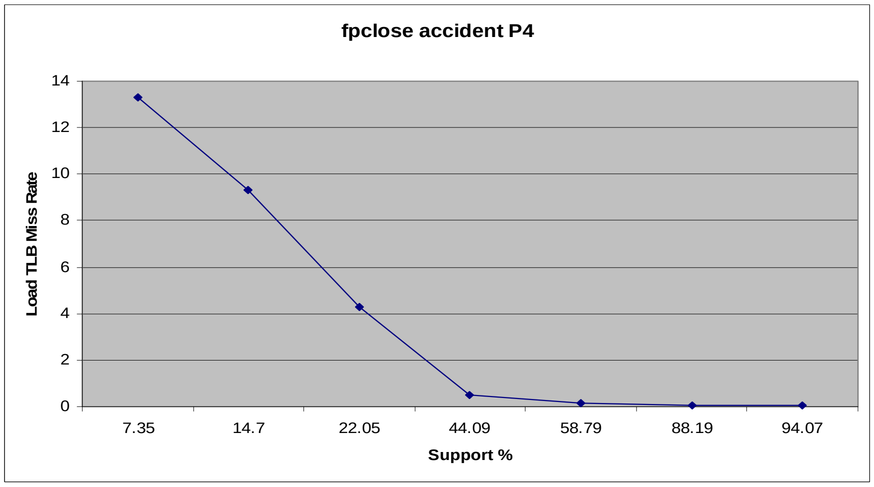


Figure 24: Data TLB Load Miss Rates (Pentium 4)

Figure 25 shows the number of closed frequent itemsets (CFIs) present in the pumsb dataset for various support thresholds. Figure 26 shows the cycles required for finding all the CFIs. Figure 27 shows the IPC for the pumsb dataset at different support thresholds. Figure 28 shows the corresponding L1 data cache and L2 unified cache global miss rates for the pumsb dataset. Figures 29-31 show the corresponding measurements for Pentium 4 system. Figure 32 shows the DTLB load miss rates for pumsb dataset on Pentium 4.

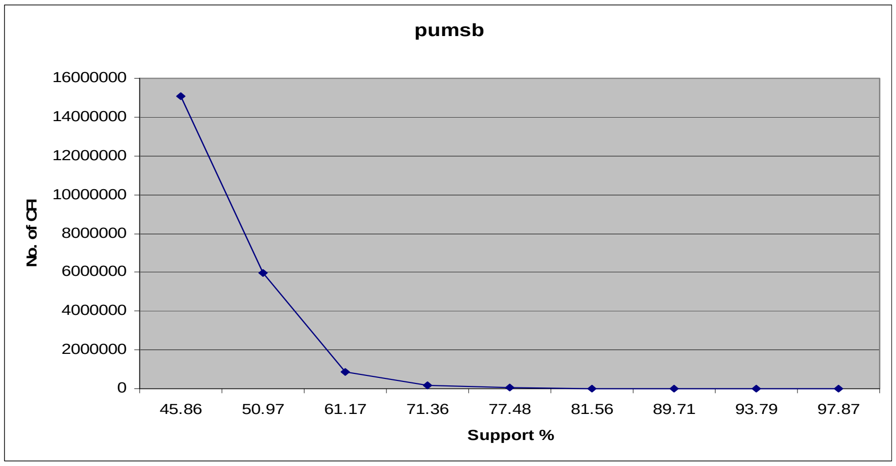


Figure 25: Distribution of Closed Frequent Itemsets

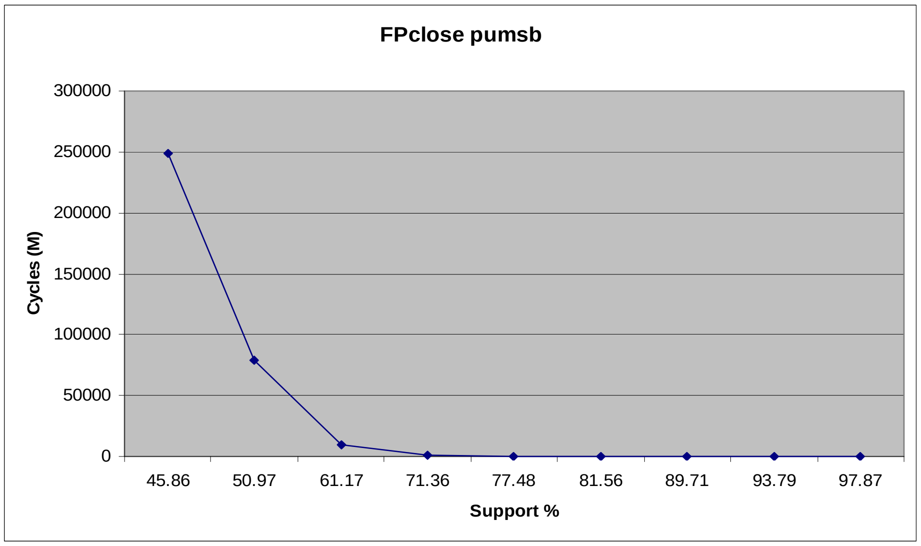


Figure 26: Execution cycles (Pentium III)

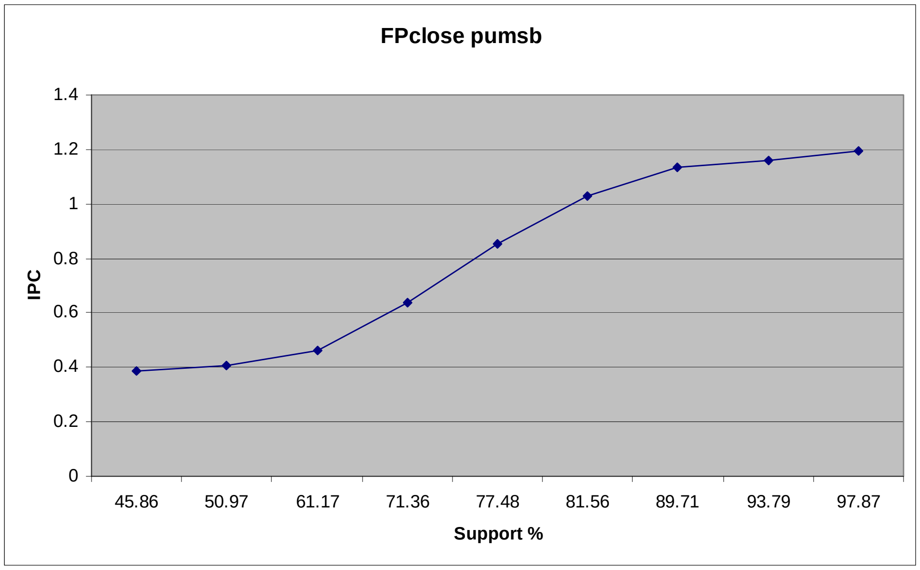


Figure 27: Instruction per Cycle (IPC) (Pentium III)

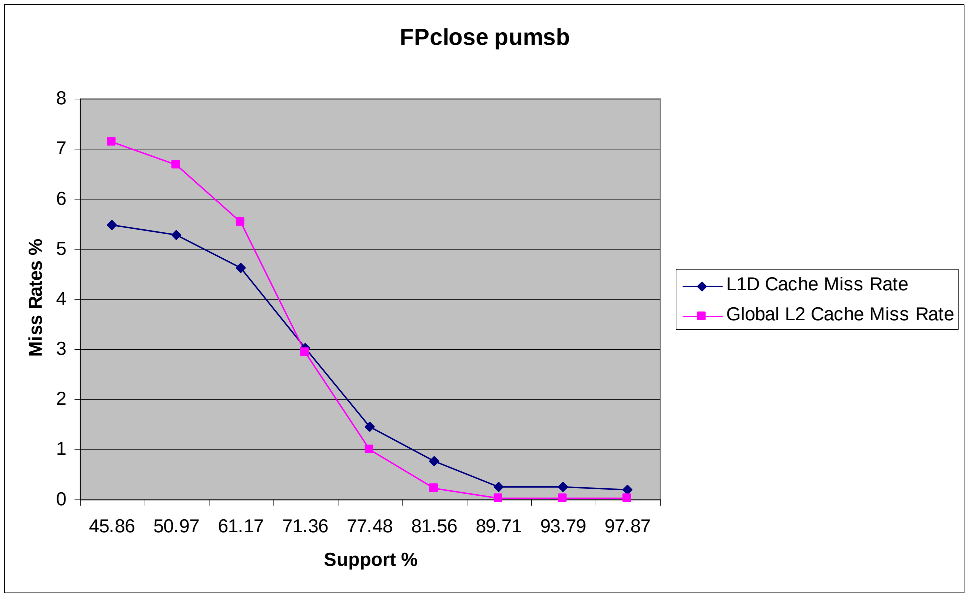


Figure 28: L1 Data and L2 Cache Miss Rates (Pentium III)

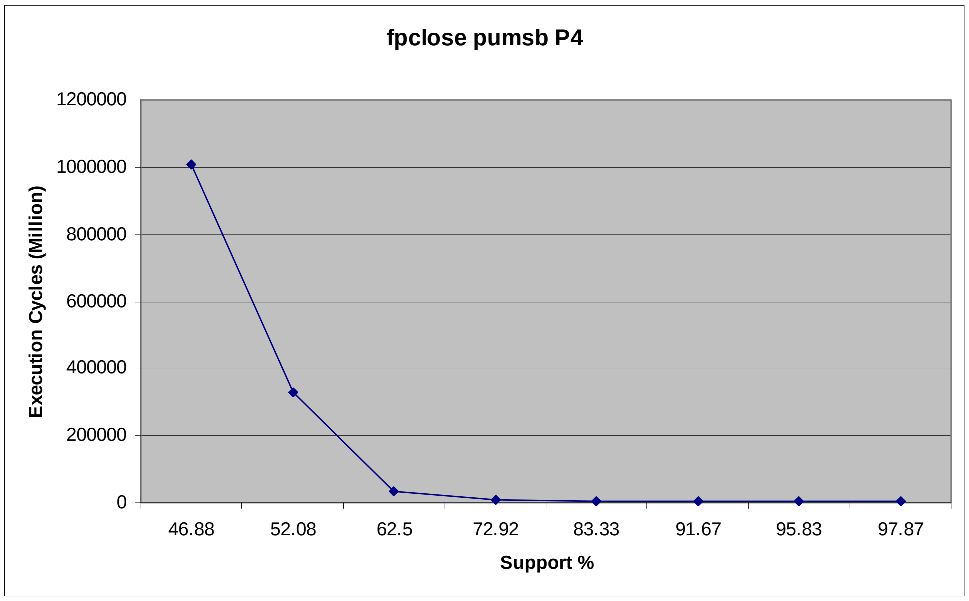


Figure 29: Execution cycles (Pentium 4)

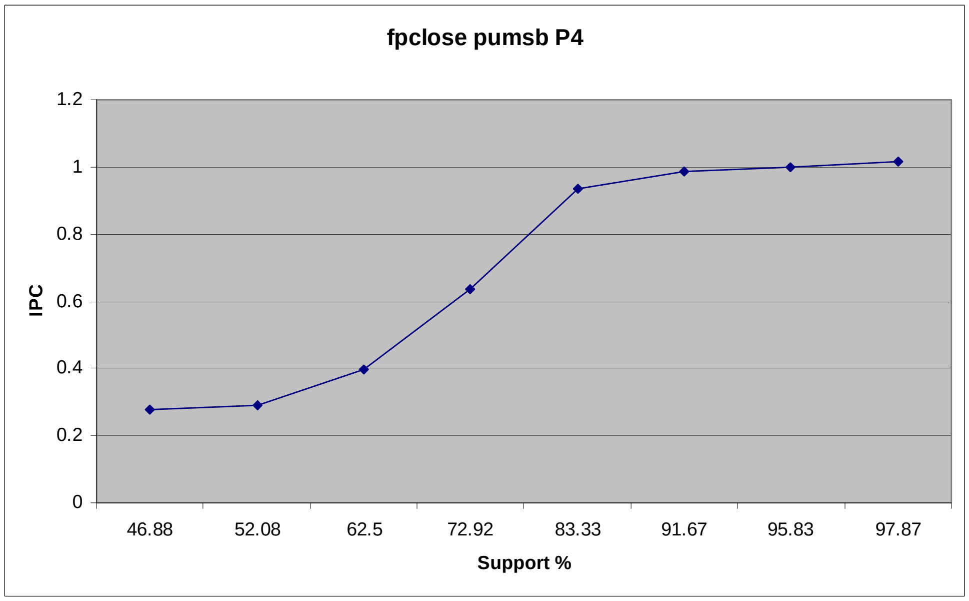


Figure 30: Instruction per Cycle (IPC) (Pentium 4)

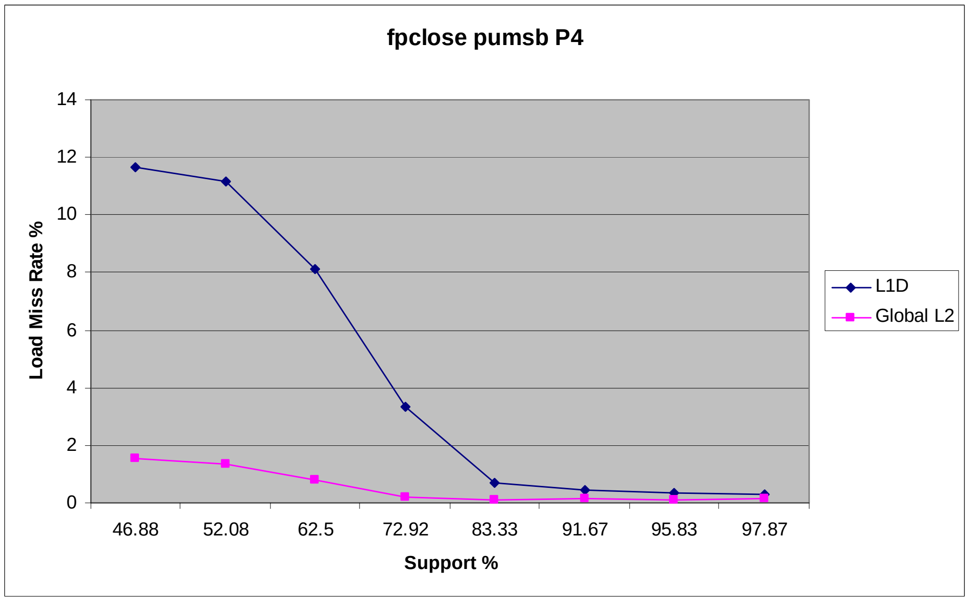


Figure 31: L1 Data and L2 Cache Load Miss Rates (Pentium 4)

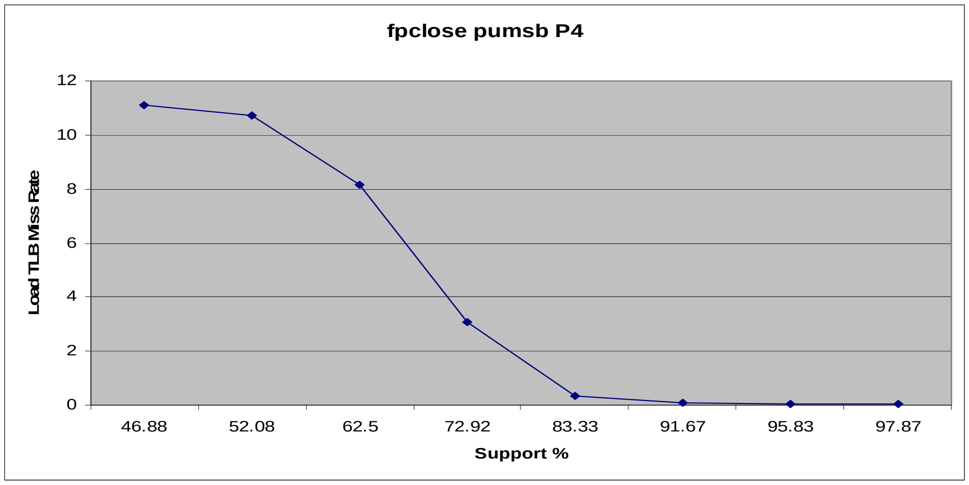


Figure 32: Data TLB Load Miss Rates (Pentium 4)

Figure 33 shows the number of closed frequent itemsets (CFIs) present in the kosarak dataset for various support thresholds. Figure 34 shows the cycles required for finding all the CFIs. Figure 35 shows the IPC for the kosarak dataset at different support thresholds. Figure 36 shows the corresponding L1 data cache and L2 unified cache global miss rates for the kosarak dataset. Figures 37-39 show the corresponding measurements for Pentium 4 system. Figure 40 shows the DTLB load miss rates for kosarak dataset on Pentium 4.

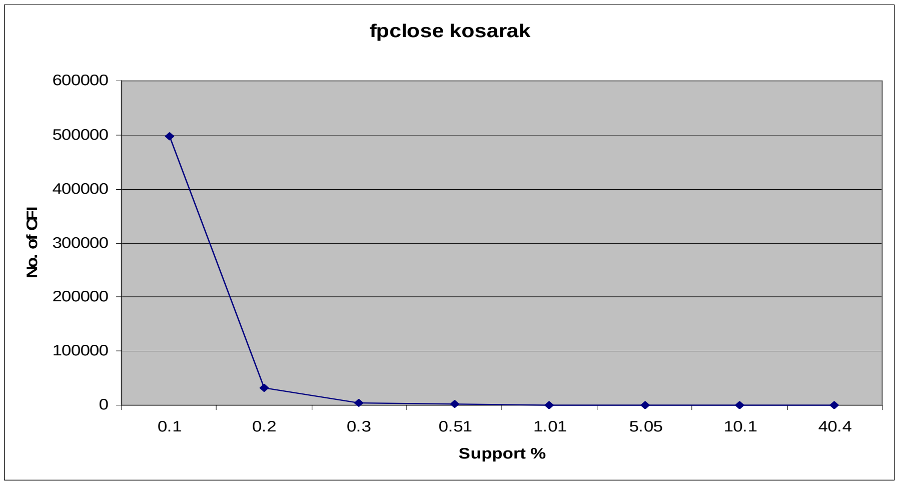


Figure 33: Distribution of Closed Frequent Itemsets

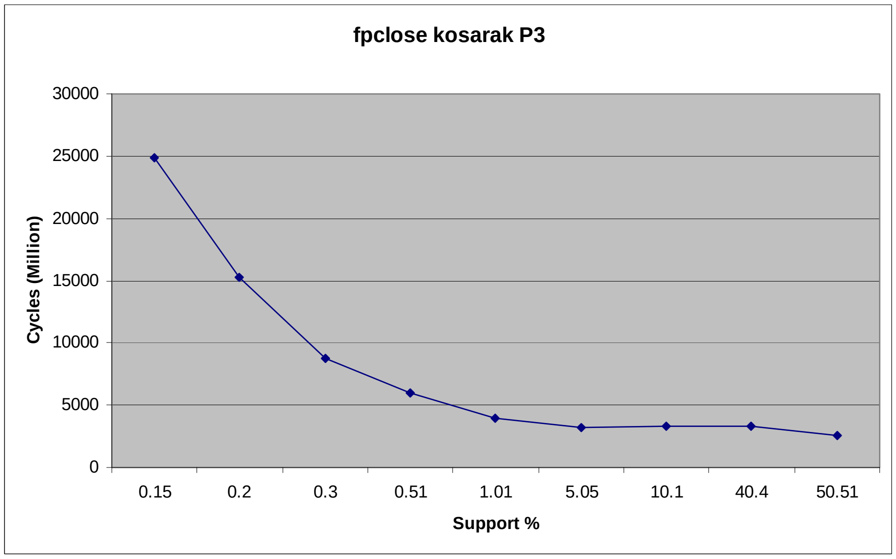


Figure 34: Execution cycles (Pentium III)

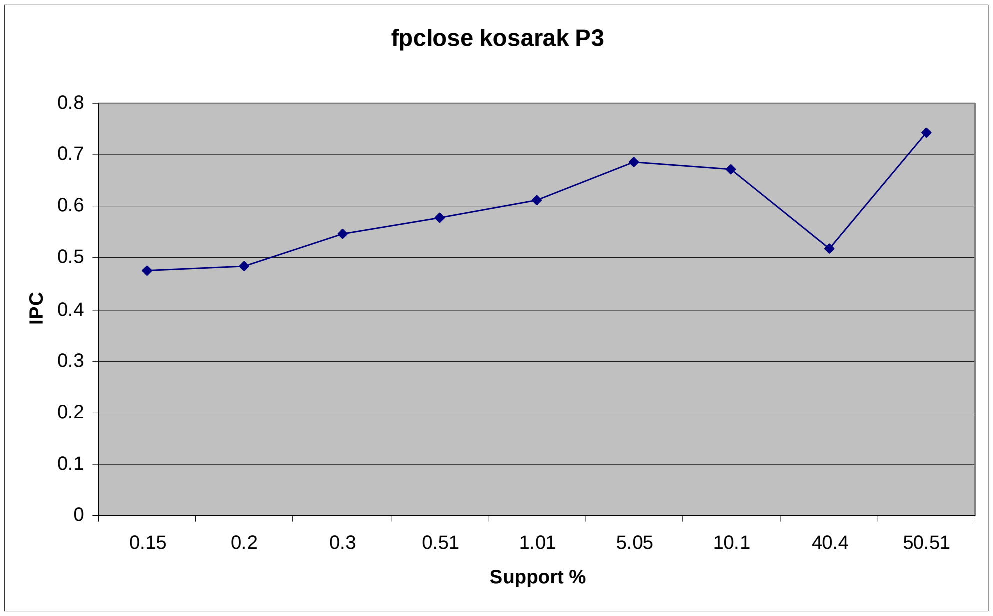


Figure 35: Instruction per Cycle (IPC) (Pentium 4)

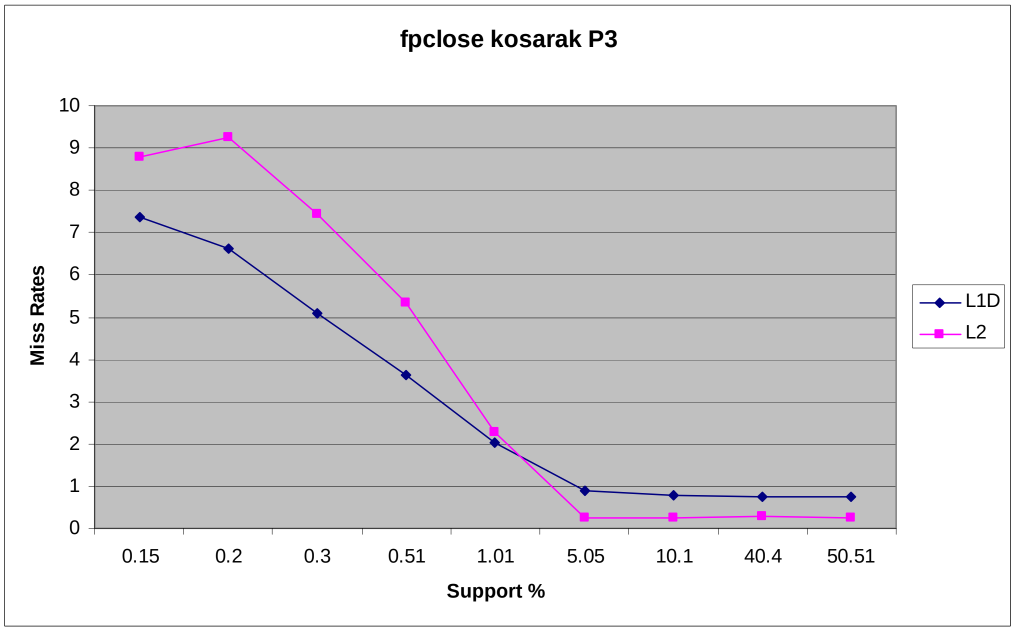


Figure 36: L1 Data and L2 Cache Miss Rates (Pentium III)

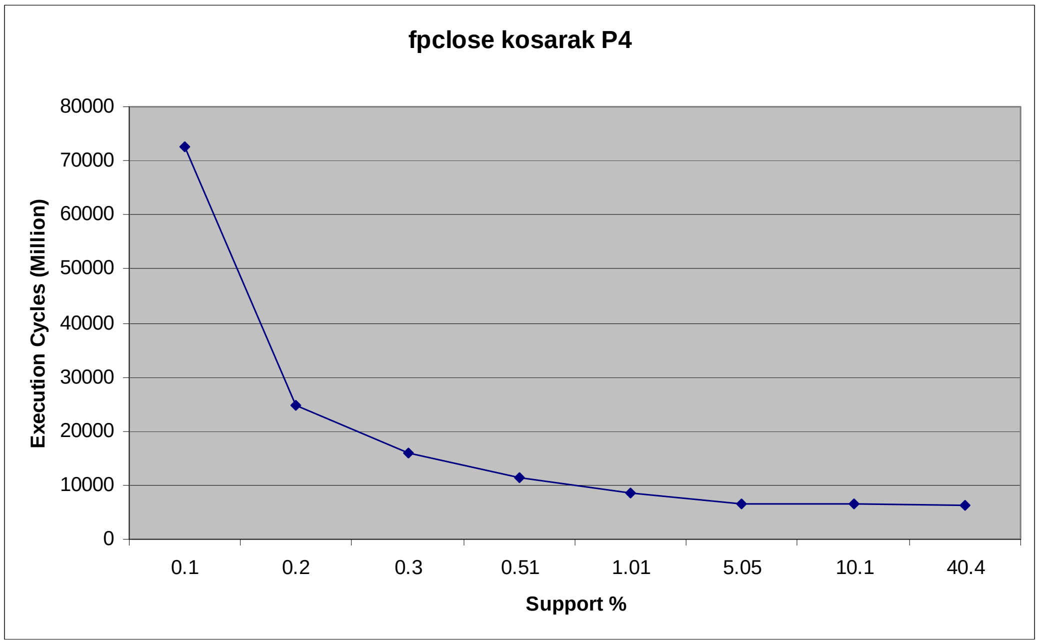


Figure 37: Execution cycles (Pentium 4)

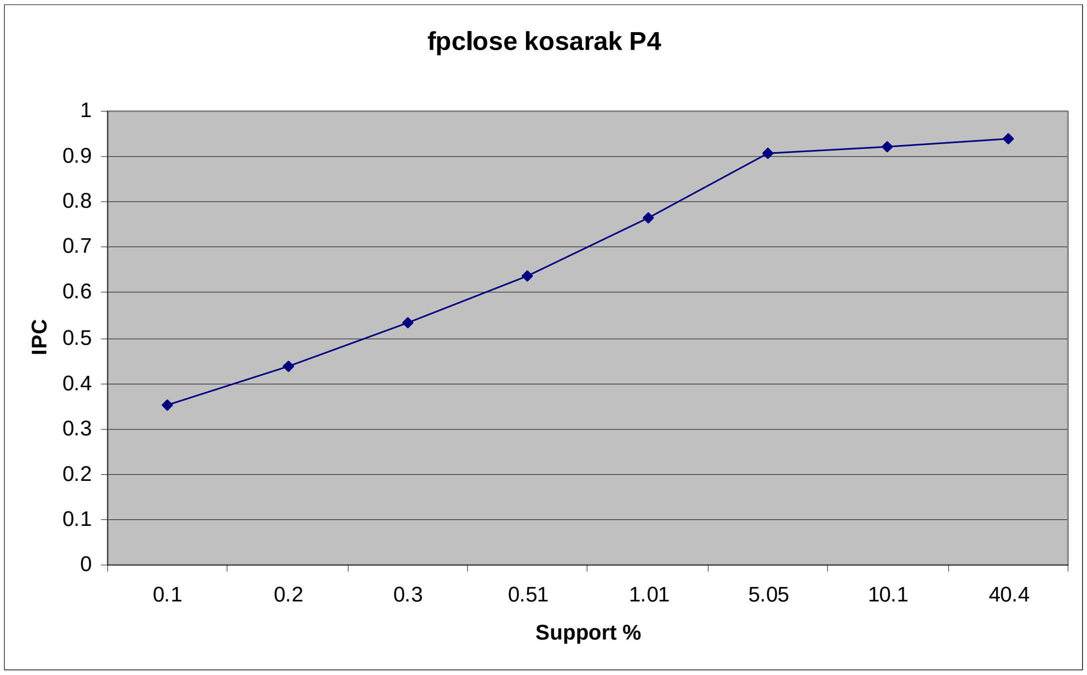


Figure 38: Instruction per Cycle (IPC) (Pentium 4)

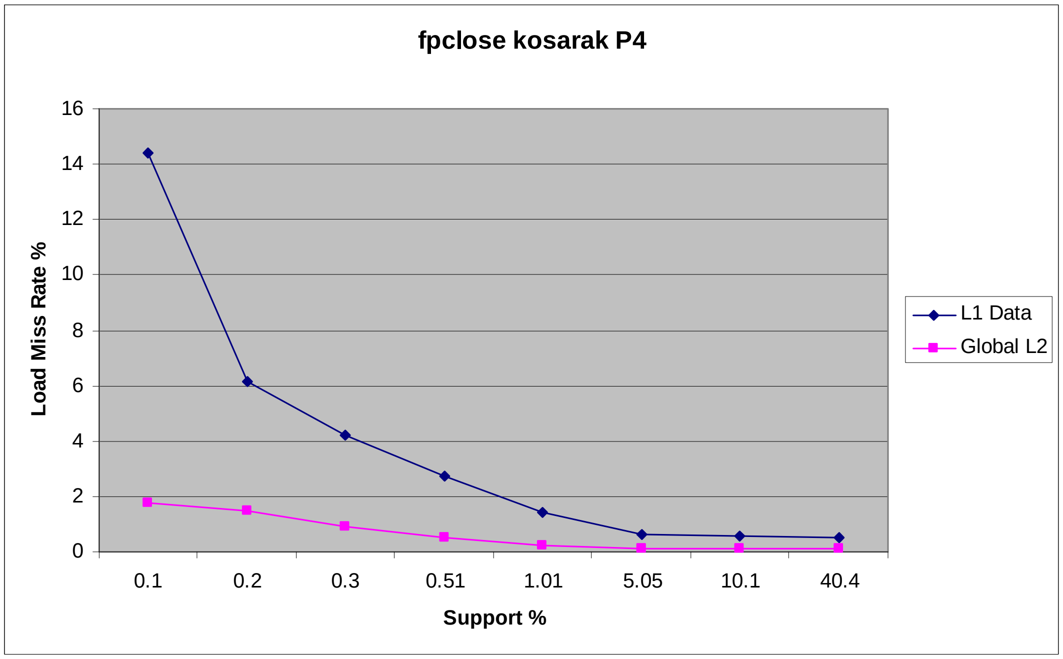


Figure 39: L1 Data and L2 Cache Load Miss Rates (Pentium 4)

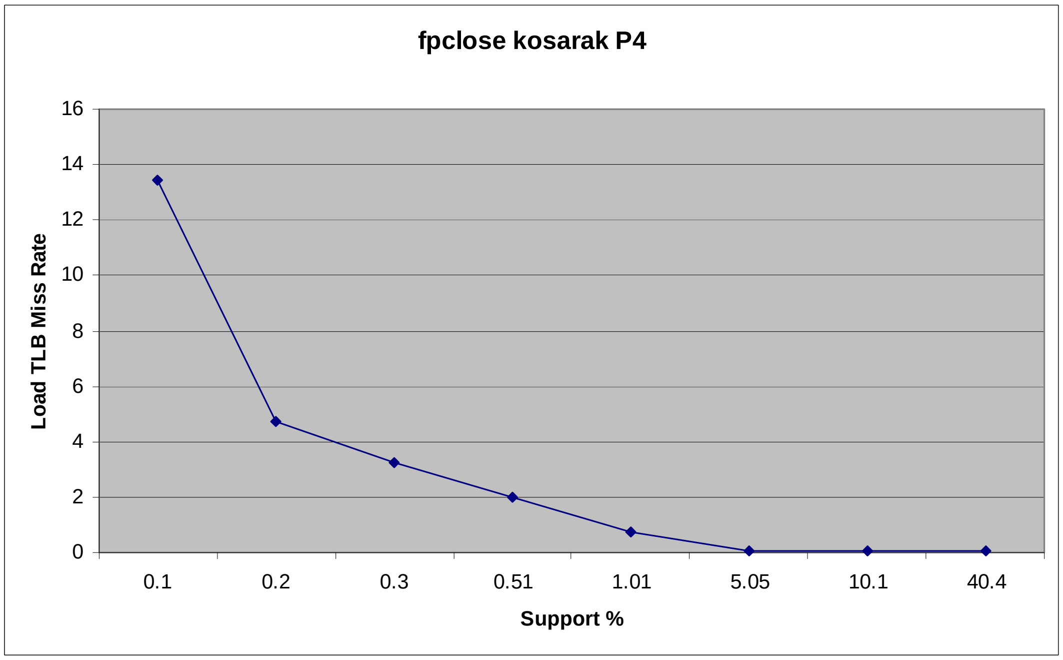


Figure 40: Data TLB Load Miss Rates (Pentium 4)

From the performance characterization we observe that execution time is proportional to the number of closed frequent itemsets (CFIs) and that the number of CFIs increases exponentially with decreasing support level. This is consistent with known algorithmic analysis of CFI mining [19]. These results clearly show that the number of CFIs is the primary determinant of execution time for FPclose. However, we also observe that IPC decreases with decreasing support level, hence IPC decreases with an increase in the number of CFIs. This reveals that as support levels decrease, the number of instructions increases exponentially and that, on average, each of these instructions takes longer to execute. For example, the IPC for the retail dataset decreases from approximately 0.75 at 0.57% support to 0.21 at 0.01% support. A similar decrease in IPC exists for all the datasets, although the dense datasets experience it at higher support levels. We also note that most datasets achieve an IPC close to or above 1 for very high support levels, for example retail achieves IPC of 1.2. Therefore, at low support levels instructions can take, on average, up to six times longer to execute than at higher support levels.

The decrease in IPC is explained by examining the memory hierarchy performance of FPclose. By examining the cache and TLB miss rates, we see that memory hierarchy performance is decreasing (miss rate increases) as support level decreases. We also note that from profiling on the Pentium 4, the data TLB misses are almost proportional to the L1 data cache misses. The large number of CFIs generated at low support levels is placing more stress on the memory hierarchy, thus the average memory access time is increasing along with the total number of instructions. Furthermore, we observe that the cache and TLB miss rates start increasing at the same point as the most significant drop in the IPC (support = 0.57% for retail and 58% for accidents). To ensure that the system is not paging, we use the iostat and time utility to track page faults. For the range of support levels we examined page faults are not a performance bottleneck.

3.1.7 Symbolic Profiling

The above results indicate that FPclose's performance might be improved at low support levels by improving the memory hierarchy performance. To gain insight into the cause of increased miss rates, we enabled symbolic profiling so that we can map execution time and cache misses to specific functions in the source code [10, 13]. Figure 41 shows the cycles spent in various functions versus support thresholds for the retail dataset. For all datasets we display only functions that account for at least 5% of the cycles for any support level, all other cycles are categorized under ''other''. Figures 42 and 43 shows the fraction of cache misses. Here also for all datasets we display only functions that account for at least 5% of the misses for any support level. All the results are for a Pentium III system.

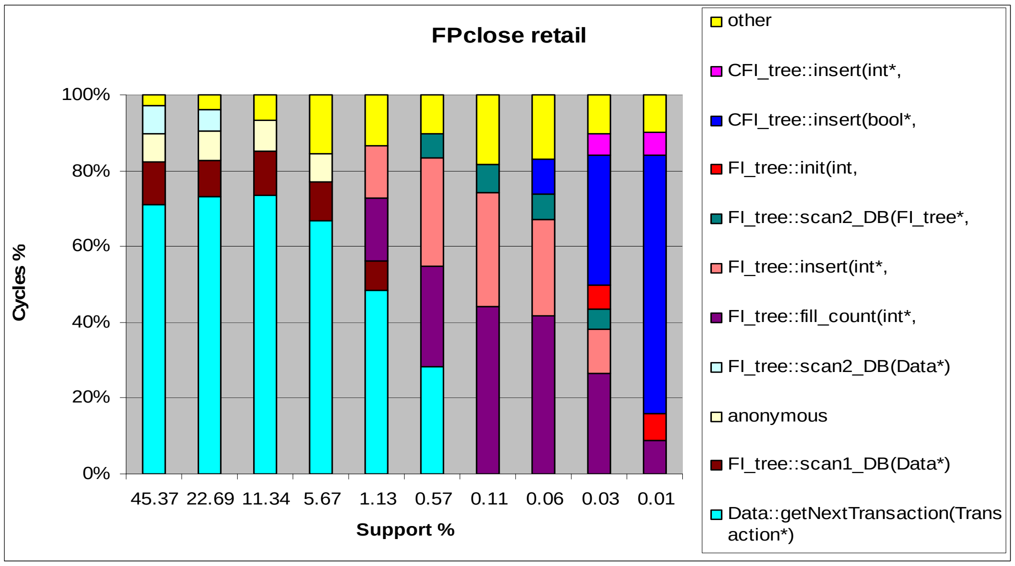


Figure 41: Execution Cycles Symbol Profile

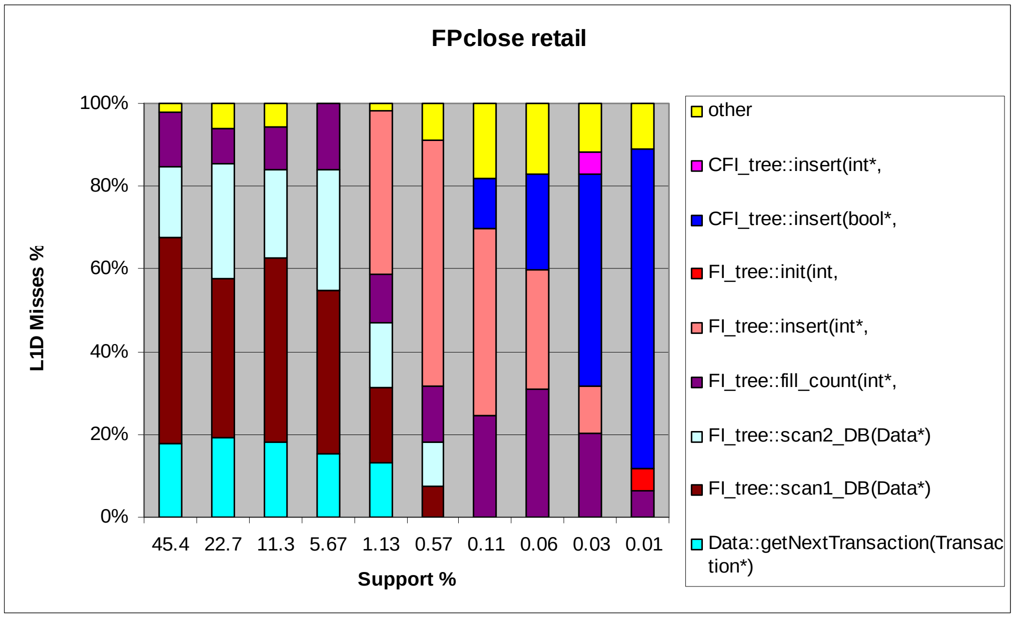


Figure 42: L1 Data Cache Misses Symbol Profile

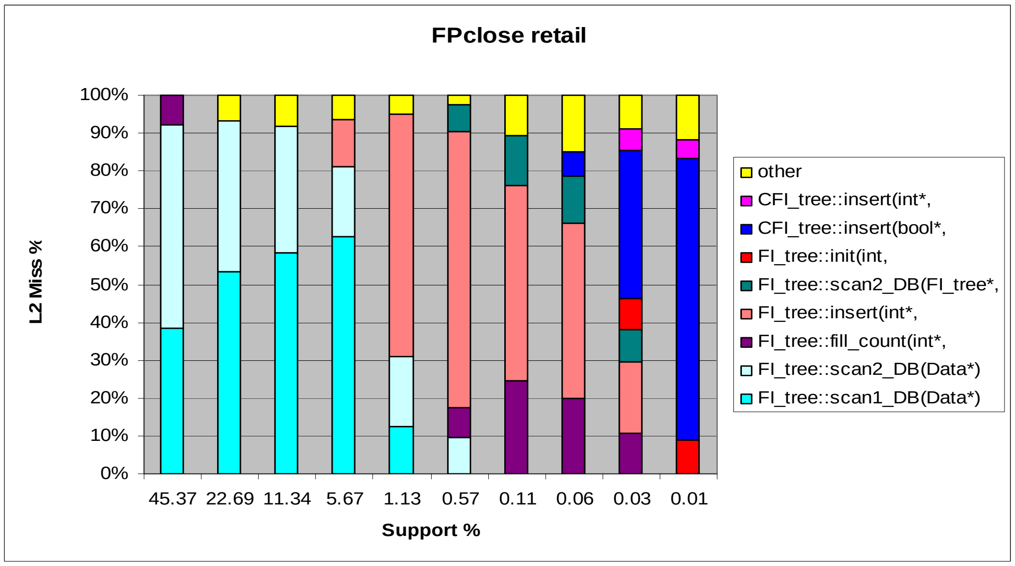


Figure 43: L2 Cache Misses Symbol Profile

Figure 44 shows the cycles spent in various functions versus support thresholds for the pumsb dataset. Figures 45 and 46 show the fraction of cache misses.

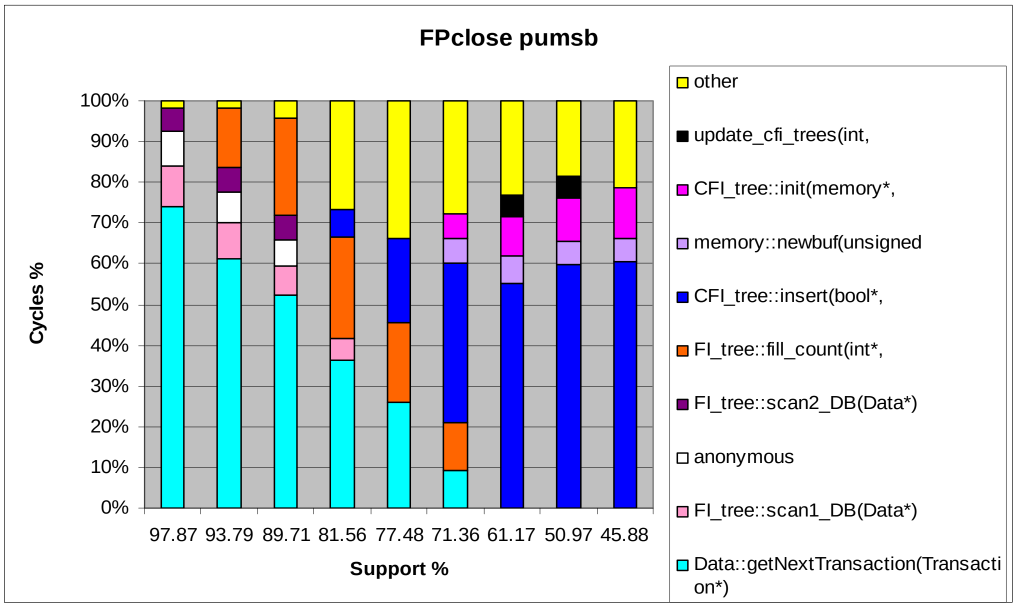


Figure 44: Execution Cycles Symbol Profile

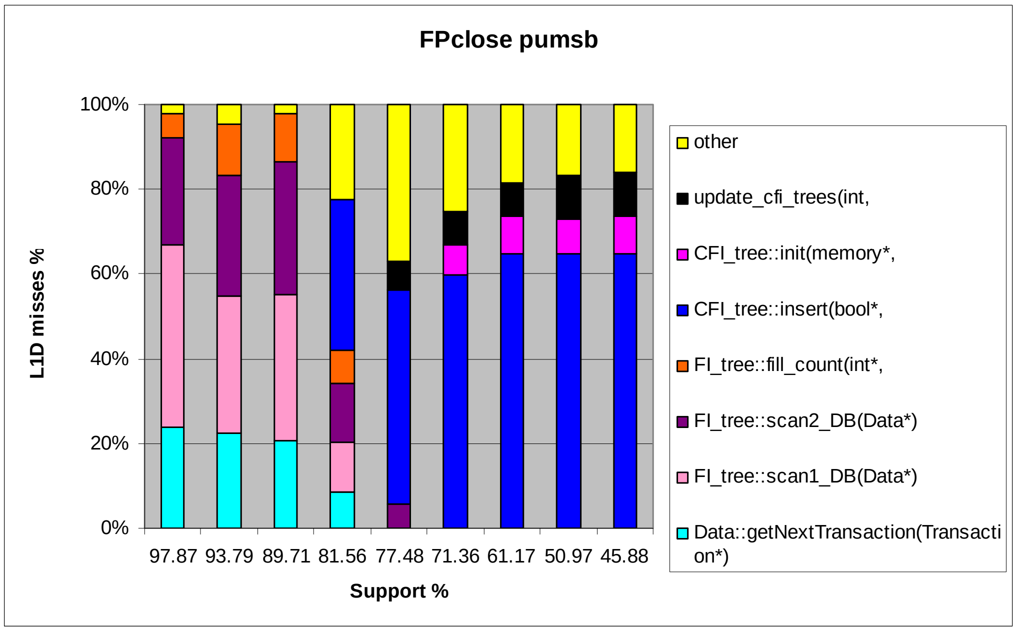


Figure 45: L1 Data Cache Misses Symbol Profile

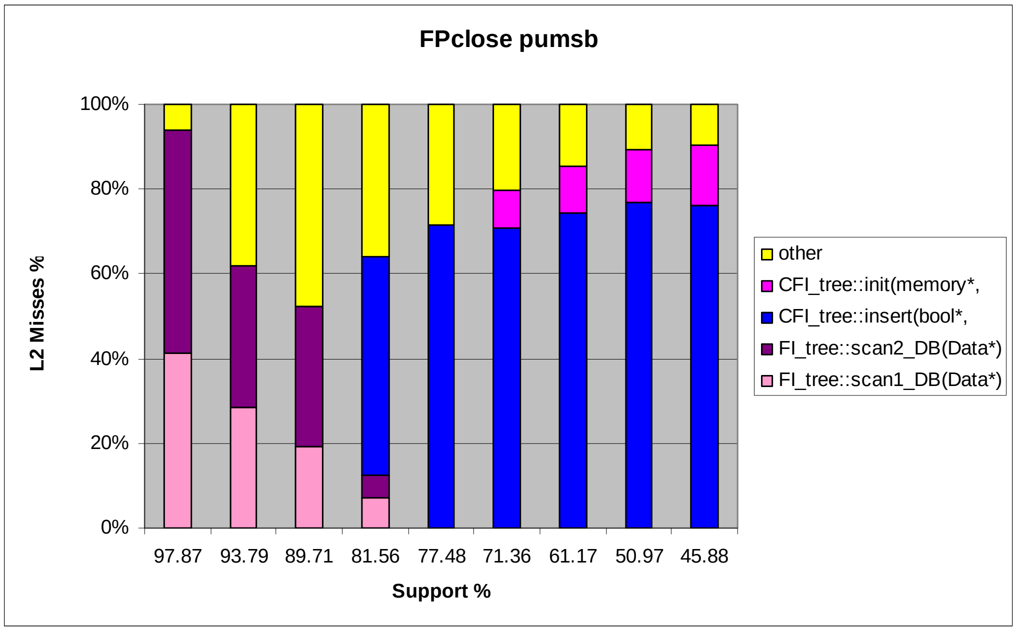


Figure 46: L2 Cache Misses Symbol Profile

Figure 47 shows the cycles spent in various functions versus support thresholds for the accident dataset. Figure 48 and 49 shows the fraction of cache misses.

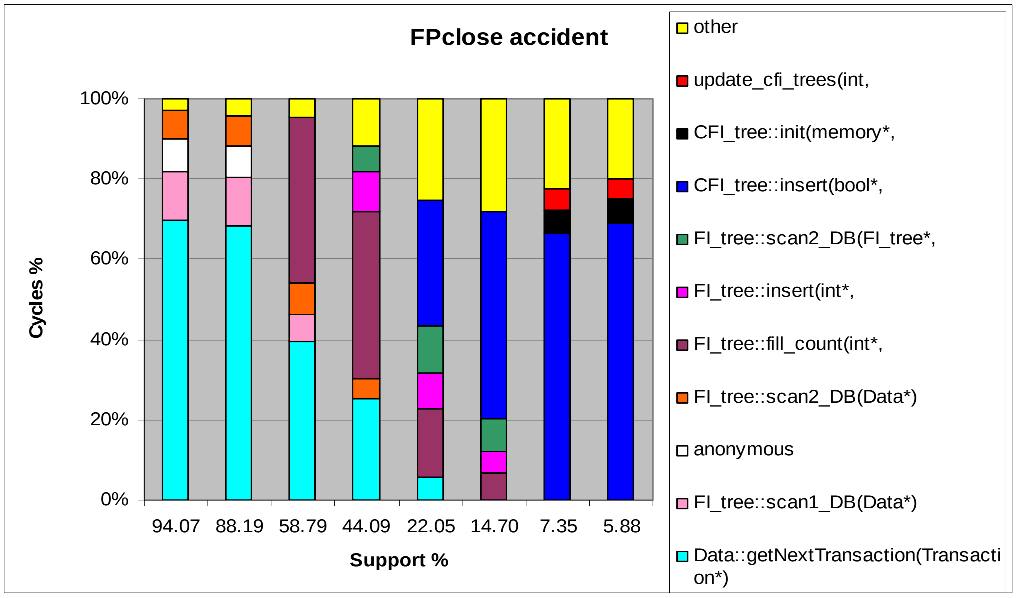


Figure 47: Execution Cycles Symbol Profile

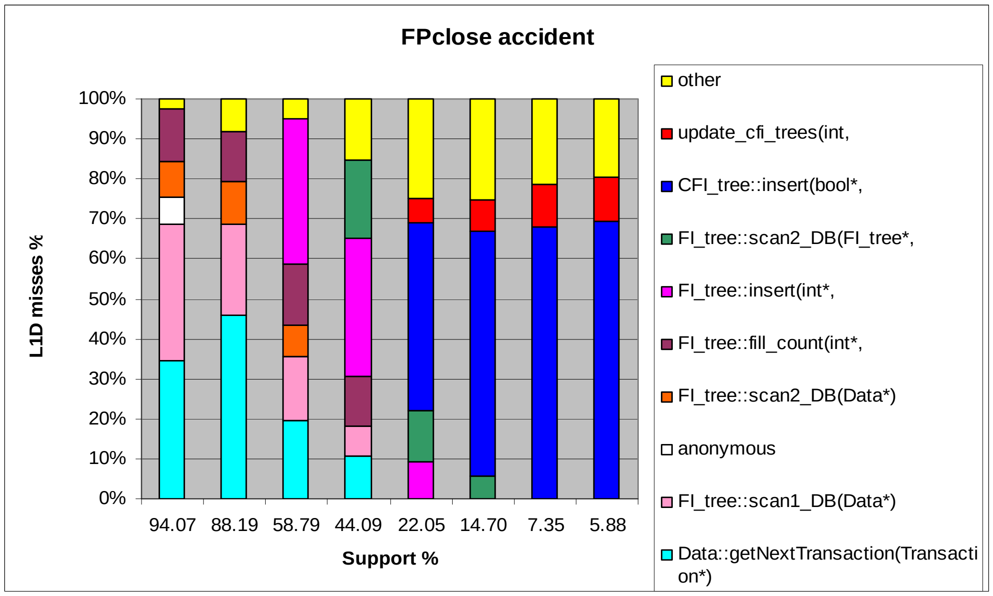


Figure 48: L1 Data Cache Misses Symbol Profile

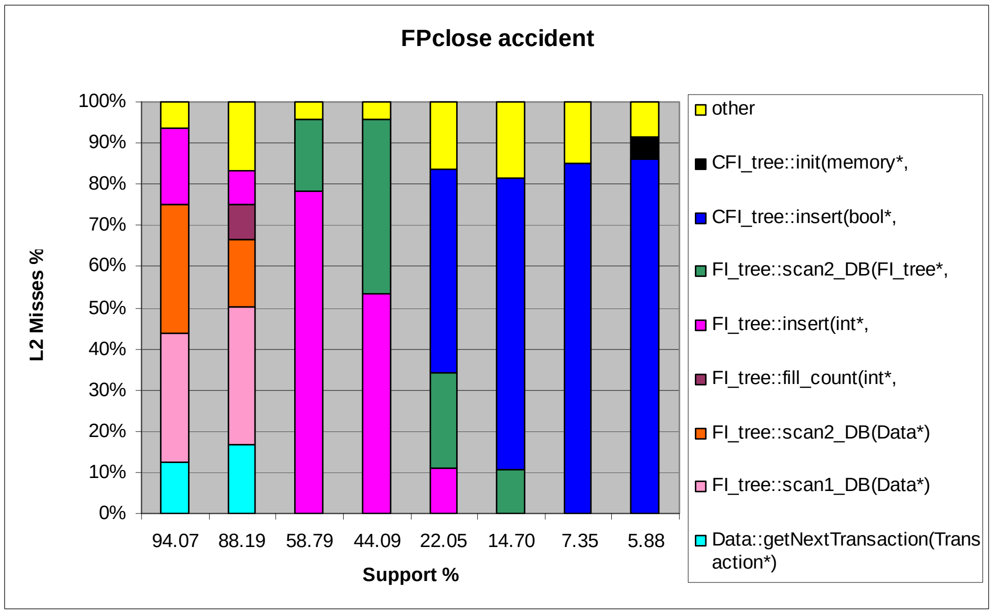


Figure 49: L2 Cache Misses Symbol Profile

These results reveal that at low support levels most of the execution time is spent in the function CFI::insert(bool*,…). A significant number of L1 data cache misses, L2 cache misses, and TLB misses also take place in this function at low support levels. All datasets reveal this same behavior: poor memory hierarchy performance leads to decreased performance in the CFI::insert(bool*,…) function.

We now analyze the performance characteristics of the insert() function in more depth. Figure 50 shows the IPC for insert() function, and compares it with the combined IPC for rest of the functions for retail dataset. We consider only those support levels at which the insert() function dominates in terms of execution cycles. Although the performance of the insert() and rest of the functions are dependent on each other to some extent, but still this differentiation gives us some idea about the relative performances. The IPC of the insert() function is lower than of rest. This brings down the IPC of the whole process. The total IPC number may be a bit different from that reported in process level profiling section, as symbolic profiling uses a slightly different executable which was compiled without optimization to preserve accurate symbol table information. Figure 51 shows the corresponding cache miss rates. L2 cache miss rates are local. Large number of misses, along with higher miss rates in most of the categories for insert() function, contributes to this difference in IPC. We ran these experiments on a Pentium III system.

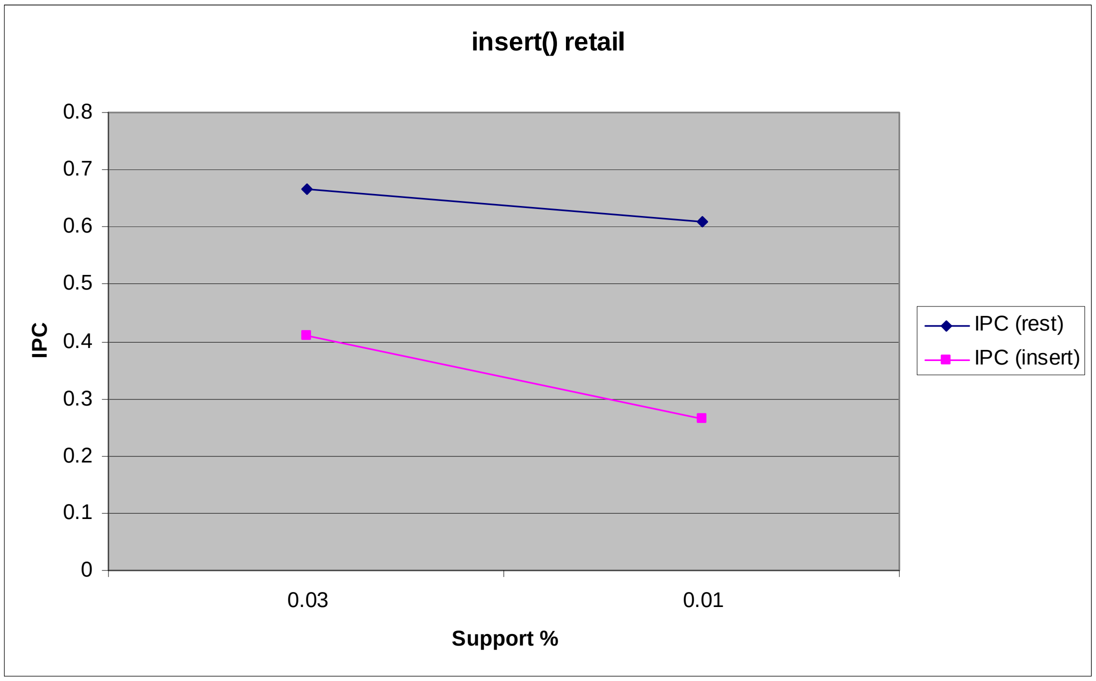


Figure 50: IPC comparison between insert() and rest of the functions

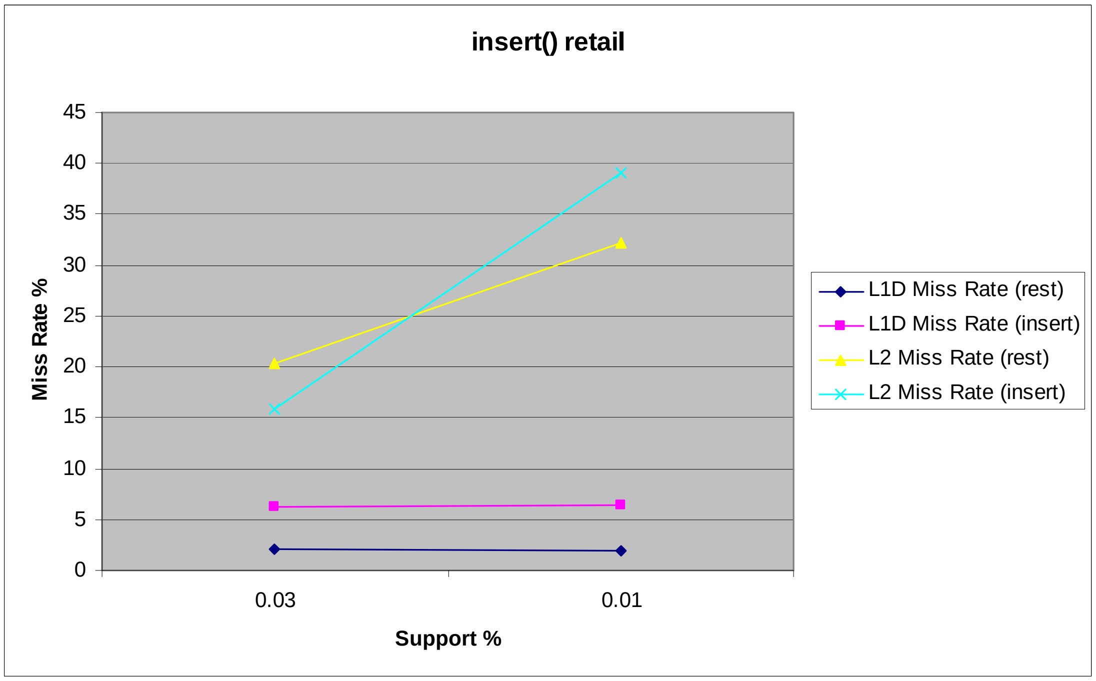


Figure 51: Miss Rate comparison between insert() and rest of the functions

Figure 52 shows the IPC comparison for pumsb dataset. Figure 53 shows the corresponding miss rate comparison. The IPC of insert() function is lower than that of the rest of the symbols primarily due to higher cache miss rates and the large number of misses.

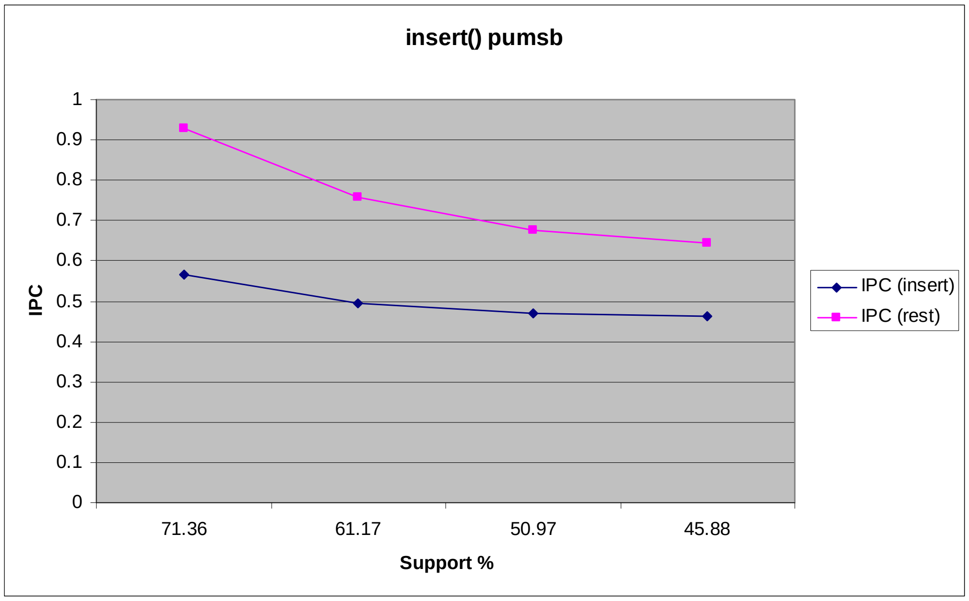


Figure 52: IPC comparison between insert() and rest of the functions

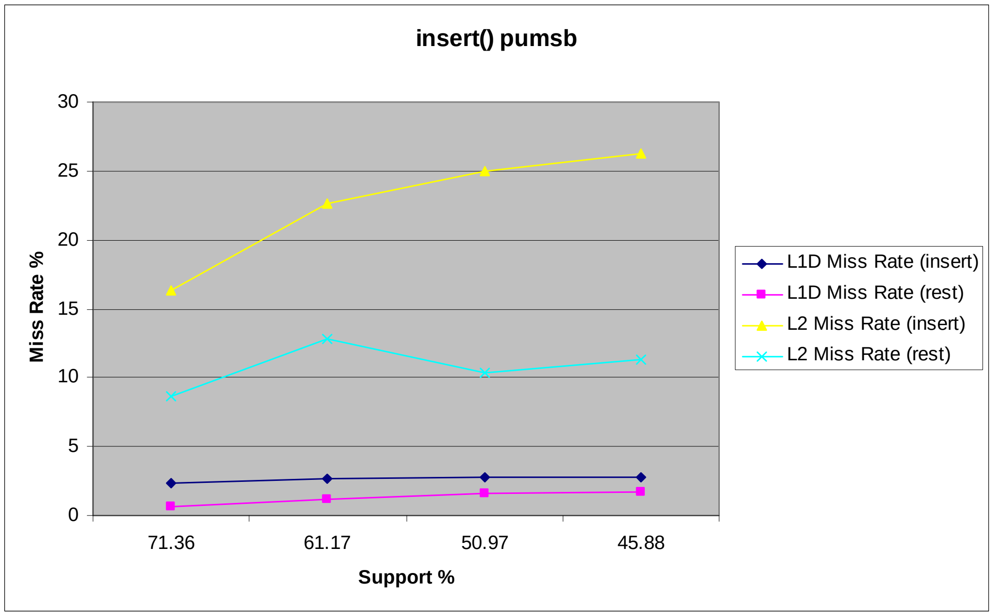


Figure 53: Miss Rate comparison between insert() and rest of the functions

From Figure 54 and Figure 55 we see a similar trend for accident dataset, though the difference between the two L2 cache miss rates seem to be much higher.

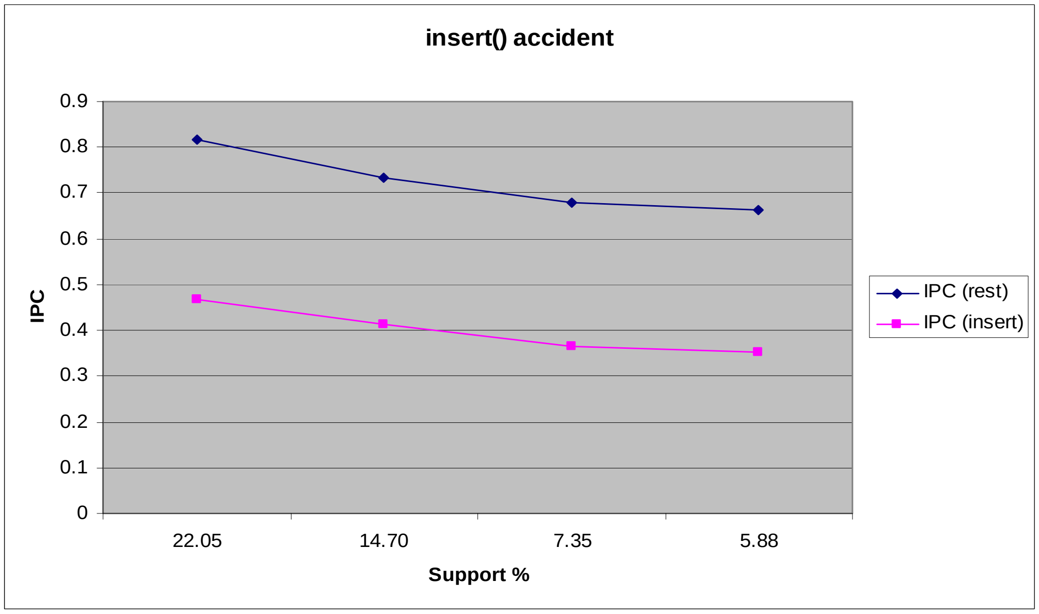


Figure 54: IPC comparison between insert() and rest of the functions

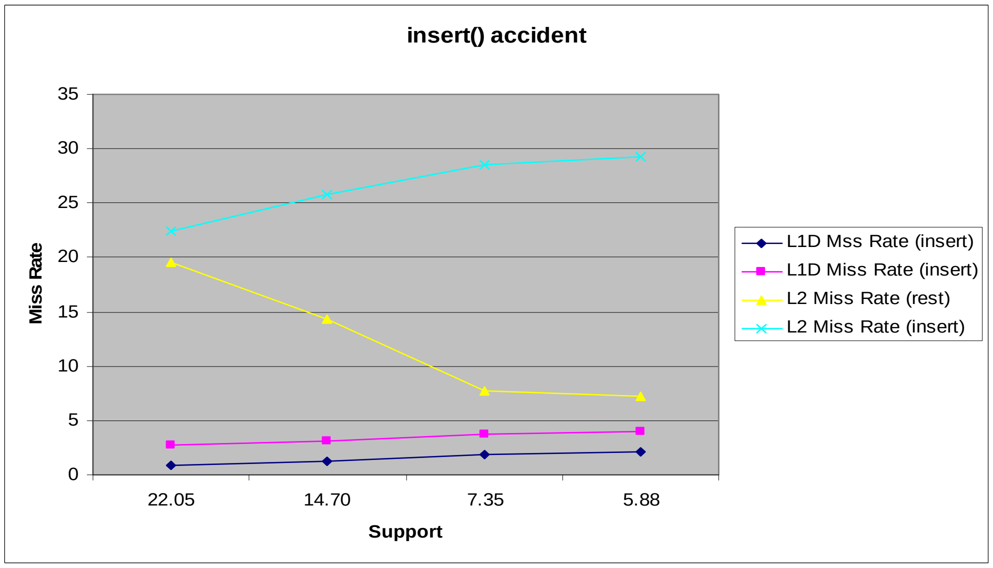


Figure 55: Miss Rate comparison between insert() and rest of the functions

CFI::insert(bool*,…) function is used for inserting itemsets in a prefix tree based data structure called the CFI-tree (Closed Frequent Itemset tree). The CFI-tree is a compact representation of all relevant frequency information in a database. Every branch of the tree represents a closed frequent itemset, and the nodes along the branches are stored in decreasing order of frequency of the corresponding items, with leaves representing the least frequent items. Compression is achieved by building the tree in such a way that overlapping itemsets share prefixes of the corresponding branches. Each node has a left child and right sibling pointer. This function is responsible for first determining if an itemset exists in the CFI tree and if not, to insert the itemset. The insertion process starts at the root. The right-sibling pointer is continuously accessed till we find a node which contains the same item as the first item in the itemset to be inserted. If there is no match, then the itemset is appended to the root. If there is a match we continue with the same procedure on the next level, appending the rest of the itemset in case of a miss. The left child pointer of the root is followed to reach the next level. Thus in order to determine if the item already exists in the CFI tree, all the items already present at the specific level of tree are examined. Since each item is its own node in the tree, the search requires extensive pointer dereferences, to access sibling nodes, that frequently result in cache and TLB misses.

This search for a sibling node which contains the required item is the bottleneck within the insert() function. Figure 56, Figure 57, and Figure 58 show that the major proportion of execution cycles, L1 data cache misses and L2 misses take place in this search for all the datasets.

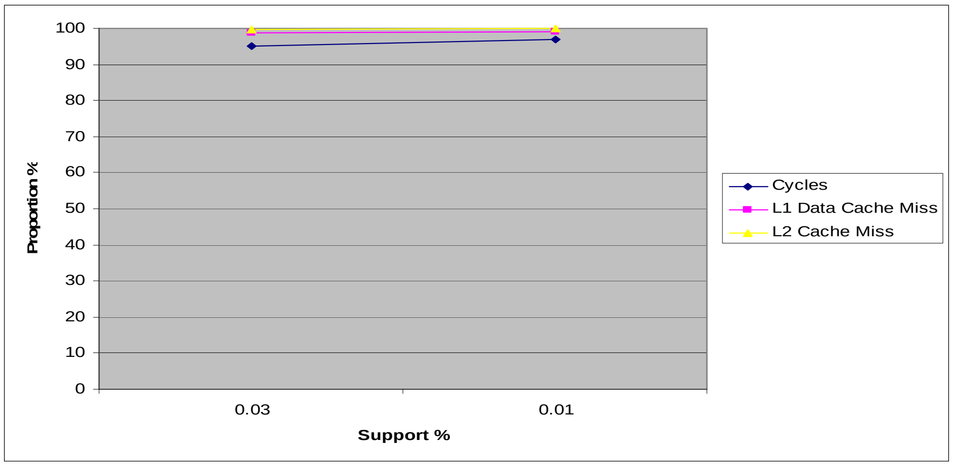


Figure 56: Proportion of cycles and misses in the search within insert for retail dataset

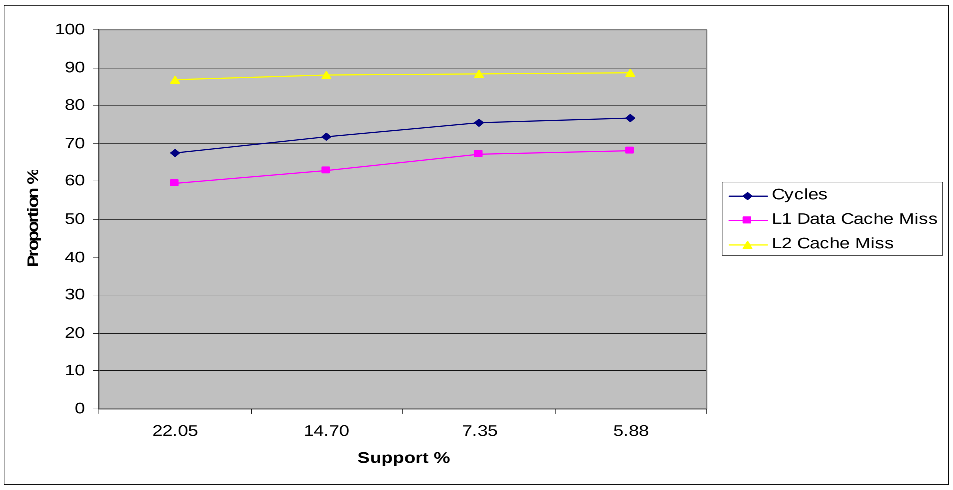


Figure 57: Proportion of cycles and misses in the search within insert for pumsb dataset

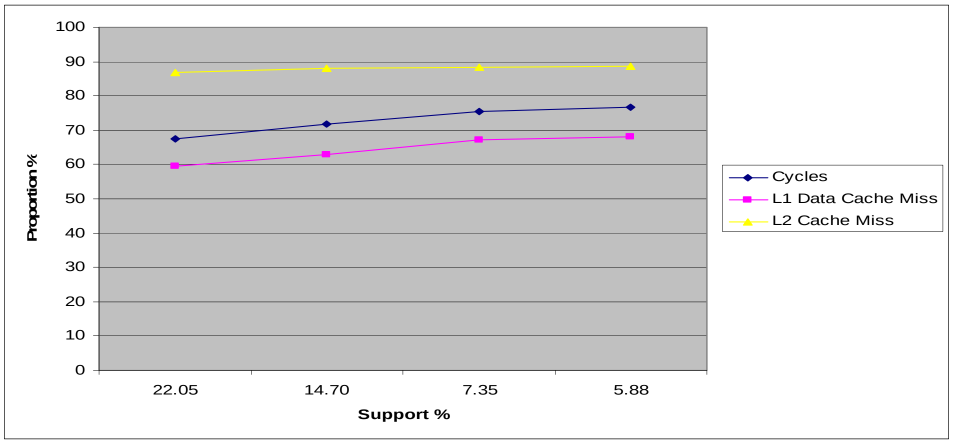


Figure 58: Proportion of cycles and misses in the search within insert for accident dataset

3.1.7.1 Simulation Analysis

In this section we present simulation results for the performance of the FPclose algorithm on a Pentium III like system with perfect memory hierarchy. Perfect memory hierarchy means that there are no cache misses. We compare it with its performance on a Pentium III like system having real memory hierarchy. We do this simulation to verify our conjecture that the drop in IPC is primarily due to large cache miss rates. Results for retail, pumsb and accident dataset are shown in Table 6. We see that large gains in performance are possible for both sparse and dense datasets if we can get the effect of a perfect memory hierarchy. This shows that IPC degrades primarily due to large cache miss rates. This result is the best case scenario for any scheme hoping to improve the performance of the memory hierarchy for the FPclose algorithm.

Table 6: SimpleScalar Simulation Results for a Perfect and Real Memory Hierarchy

Dataset	Type of Dataset	IPC (Perfect)	IPC (Real)	Support %
Retail	Sparse	2.0990	0.6236	0.03
accidents	Dense	1.1490	0.5405	14.7

We next show the impact of a wide instruction window on the performance of the algorithm. Table 7 shows the effect of instruction window with increasing widths on the performance of the algorithm for accident dataset at support of 14.7 %. RUU size of 64 was used in the default configuration. We see that instruction window width has almost no impact on IPC. Thus it is not possible to enhance the IPC by increasing the instruction window width.

Table 7: SimpleScalar Simulation Results depicting the effect of Instruction Window Width

RUU size	IPC
64	0.5405
128	0.5412
256	0.5412
512	0.5412
1024	0.5412

3.1.8 Data Structure Modifications

This section explores several techniques for improving the memory hierarchy performance of FPclose, specifically the CFI::insert(bool*,…) function. The performance characterization from the previous section clearly identifies the insert routine's memory hierarchy behavior as a performance bottleneck for these real world datasets. In this section we present various data structure modifications that attempt to improve cache performance. The support levels at which we measure the speedups obtained by different modifications are unique for each dataset. We include five different support levels for each dataset such that execution time for the base implementation of FPClose is greater than 1 second at these levels. We compare the execution cycles of each scheme with that of the original implementation to obtain speedups.

3.1.8.1 Padding and Aligning

A simple approach that can improve cache performance is to align and pad the data structure to ensure that only integer multiples of the structure can fit into a cache line [10]. This can be achieved by padding the structure with sufficient space. The benefit of this approach is that an access to the data structure, after any miss penalty, guarantees that the entire structure is now resident in the cache. If many of the fields within the data structure are "hot" then this approach can lead to higher performance because it limits the cache misses to 1 for all subsequent accesses to the data structure. Unfortunately, this technique did not produce any noticeable change in execution time.

3.1.8.2 Node Clustering

The next approach adopts a composite approach to improve cache performance by combining sub-tree clustering and pointer elimination [4]. The approach nestles frequently accessed fields from many instances of the data structure into a composite structure that can then be accessed using an implicit address calculation. Children of a CFI tree node are clustered into a composite node by grouping their unique attributes (e.g., item, left-child, etc.) in to large arrays. The attribute arrays are indexed according to the most commonly chased pointer, right-sibling, thus eliminating the use of dereferenced pointers in favor of an incremental offset calculation. Composite nodes are allocated to hold either a static or dynamic number of nodes and can be chained together to accommodate dynamic trees in either case. This approach improves performance because frequent right-sibling chases are reduced to incremental address calculations and item comparisons. Implicit prefetching occurs because the arrays are contiguous in memory and any dedicated prefetch hardware will bring prescient portions of the arrays into the cache prior to their access.

Figure 59 shows the performance improvement for each benchmark with various composite node sizes (i.e. number of nodes combined into a single composite) for retail dataset. Figure 60 shows the same for bmspos, Figure 61 for accident and Figure 62 for kosarak. These improvements are on a Pentium III system. Figures 63-66 show the corresponding results for a Pentium 4 system.

We achieve performance gains of up to 79% on Pentium III systems and 65% on Pentium 4.

We observe that on a Pentium III system we get speedups for all the datasets across most of the support levels. The gains are lesser for dense datasets as compared to sparse ones. This is mainly due to smaller right sibling chains for dense datasets compared to sparser ones. Larger composite node sizes in such cases cause wastage of memory and cache pollution. For Pentium 4 the gains are lesser for sparser datasets compared to Pentium III results and we incur losses in majority of the cases for dense datasets. This is primarily because of larger caches in Pentium 4 which improves the performance of the base case by reducing miss rates.

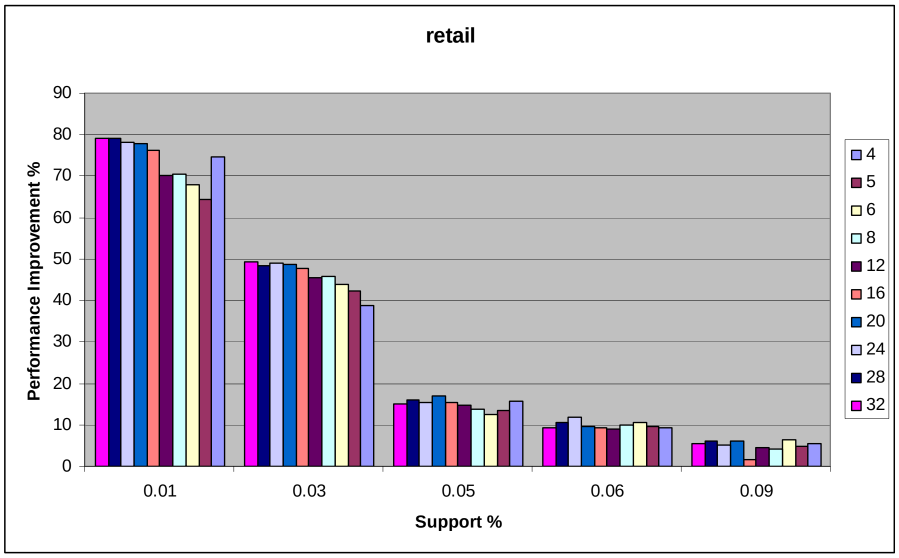


Figure 59: Performance Improvement for different composite node sizes for retail dataset

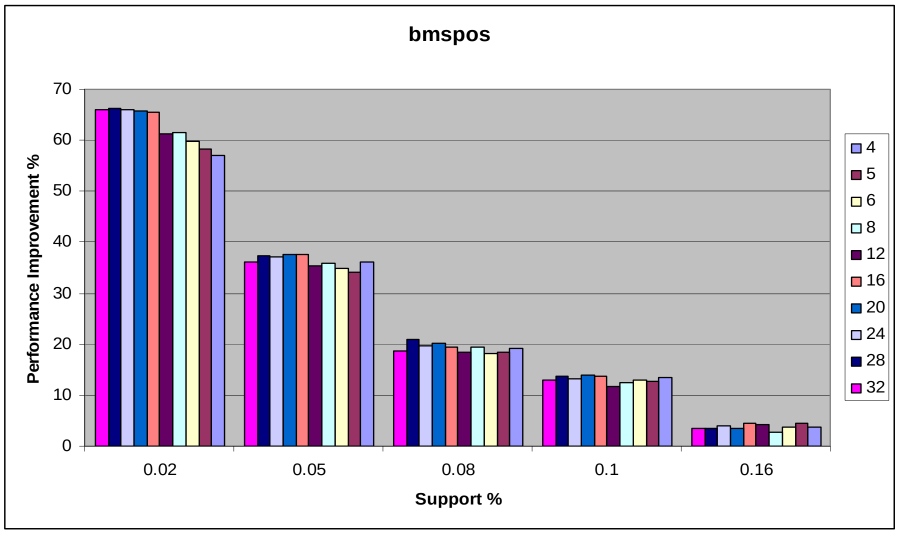


Figure 60: Performance Improvement for different composite node sizes for bmspos dataset

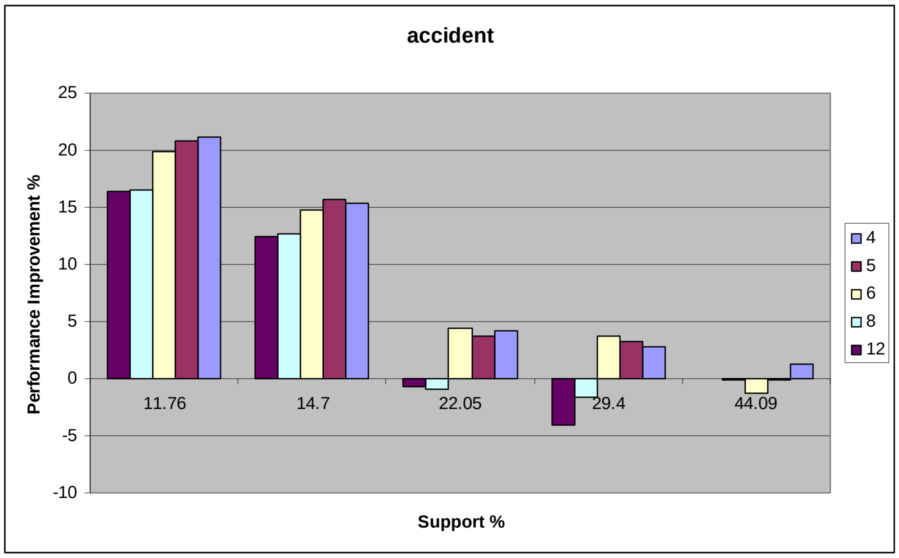


Figure 61: Performance Improvement for different composite node sizes for accident dataset

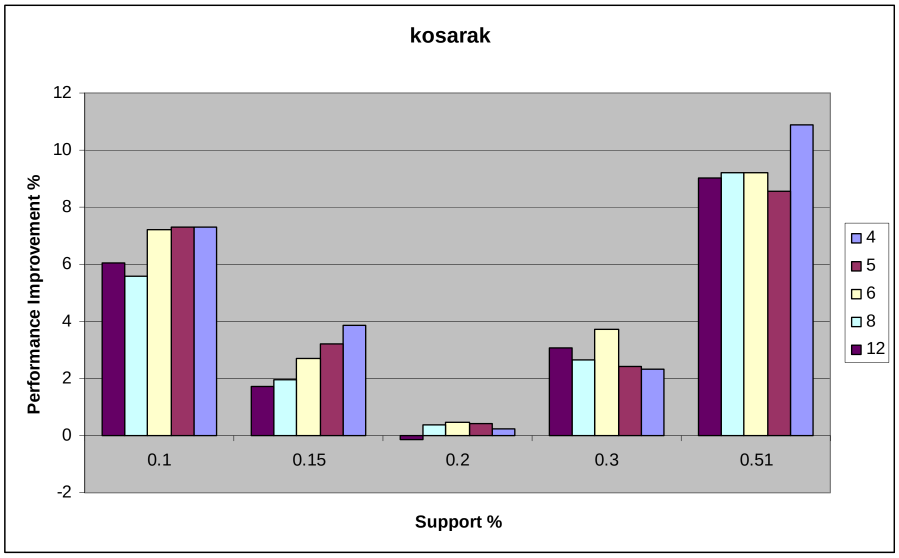


Figure 62: Performance Improvement for different composite node sizes for kosarak dataset

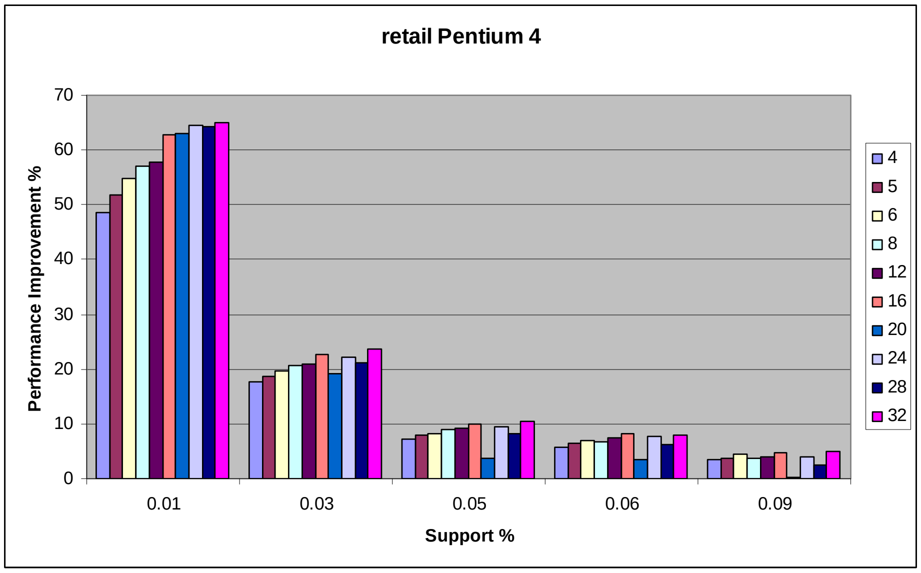


Figure 63: Performance Improvement for different composite node sizes for retail dataset

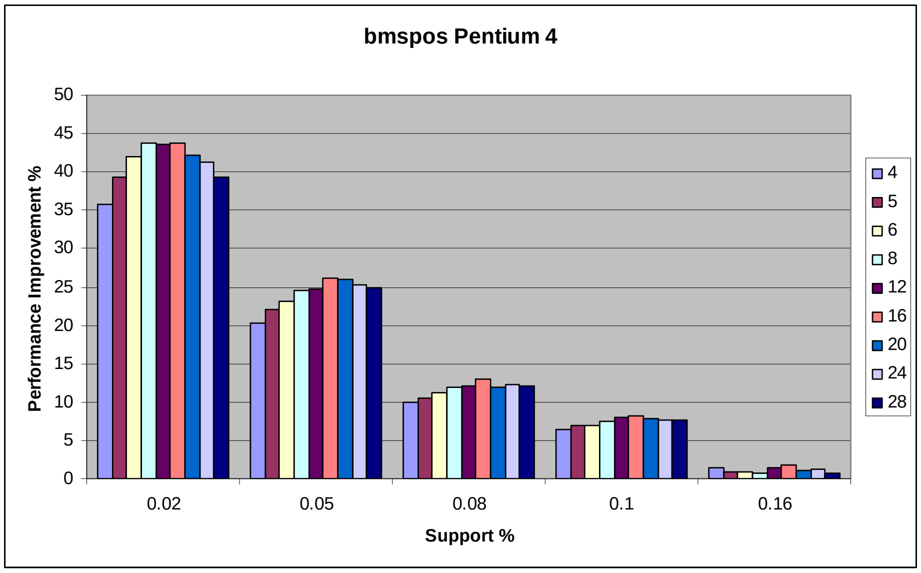


Figure 64: Performance Improvement for different composite node sizes for bmspos dataset

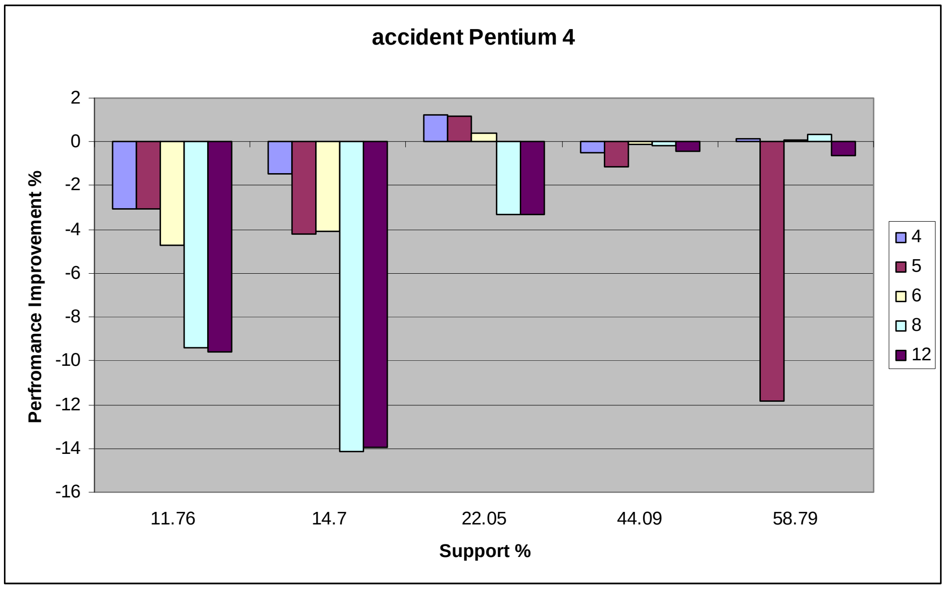


Figure 65: Performance Improvement for different composite node sizes for accident dataset

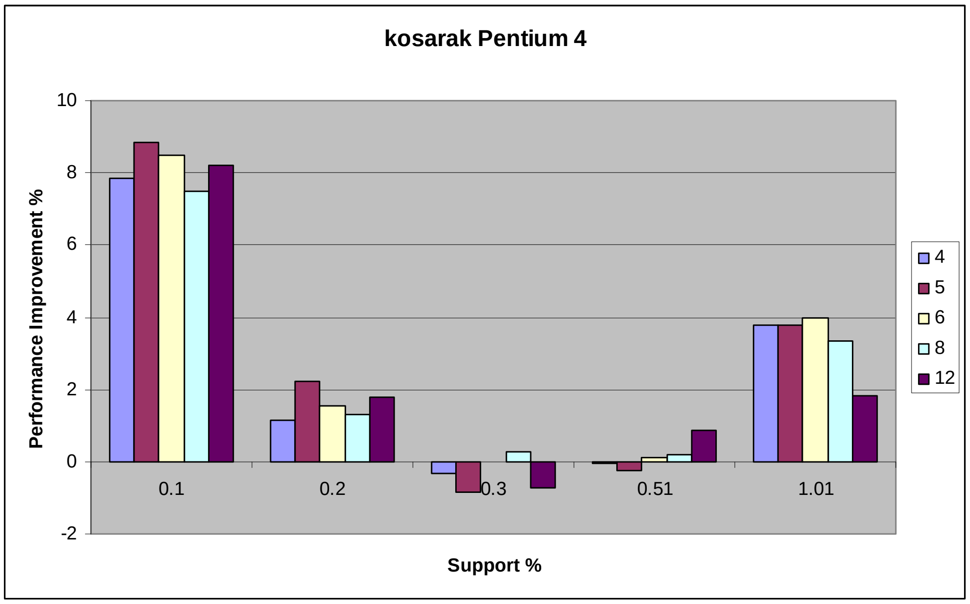


Figure 66: Performance Improvement for different composite node sizes for kosarak dataset.

These results reveal that for any single dataset one particular node cluster size does not give the maximum gains across all support levels. Also different datasets require different node clustering sizes to achieve highest gains across a range of support levels, say lower support levels.

3.1.9 Dynamic Cluster Sizing

The above results clearly show that node clustering can improve CFI mining. Unfortunately, they also reveal that each dataset requires a different cluster size to achieve maximum performance improvement. We did some characterization tests to learn that the average number of nodes accessed during the sibling traversal is data dependent. Therefore, we develop a dynamic node clustering algorithm that automatically adjusts the number of nodes clustered during allocation to be as close as possible to the length of the right sibling chain. With this scheme we try to capture the variation in the length of the right sibling chain.

The scheme we developed chooses the size of the node to be allocated to be double the size of the last node in that right sibling chain. The newly allocated node becomes the new last node in the right sibling chain and is added to the end of the chain. This has an effect of increasing the size of the composite nodes at levels in the tree where more nodes are being allocated. Figures 67-70 show the performance improvement of the dynamic scheme for retail, bmspos, kosarak, and accident dataset respectively on a Pentium III system. Figures 71-74 show the corresponding gains on a Pentium 4 system. We obtain speedups up to 81% on Pentium III system and 67% on Pentium 4. We observe the same trend as was visible for the static scheme; gains are higher on a Pentium III system as compared to Pentium 4 and gains are larger for sparse datasets. On Pentium 4 we do not observe any gains for dense datasets. This is due to the combined effect of smaller right sibling chains for dense datasets along with larger caches in Pentium 4.

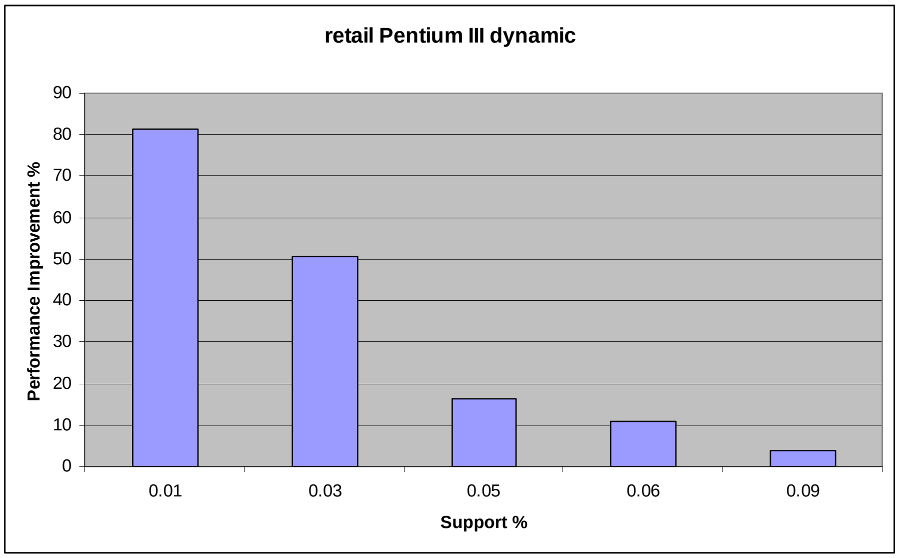


Figure 67: Performance Improvement of dynamic scheme for retail dataset on Pentium III

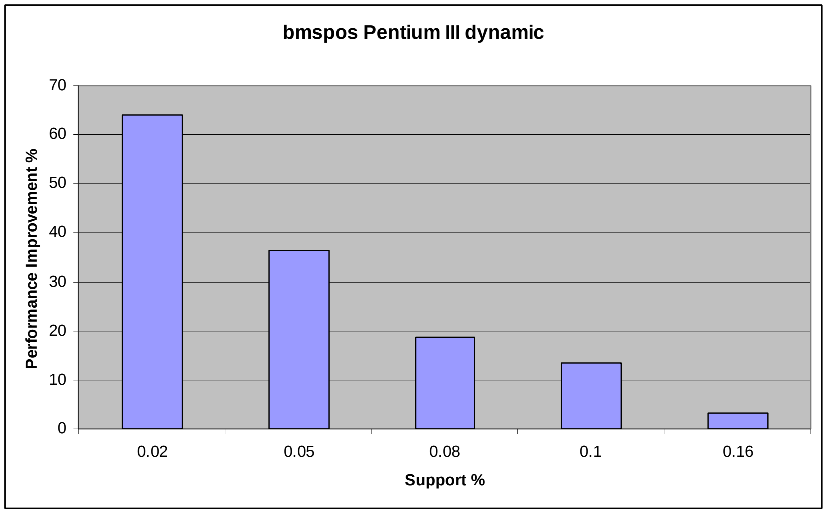


Figure 68: Performance Improvement of dynamic scheme for bmspos dataset on Pentium III

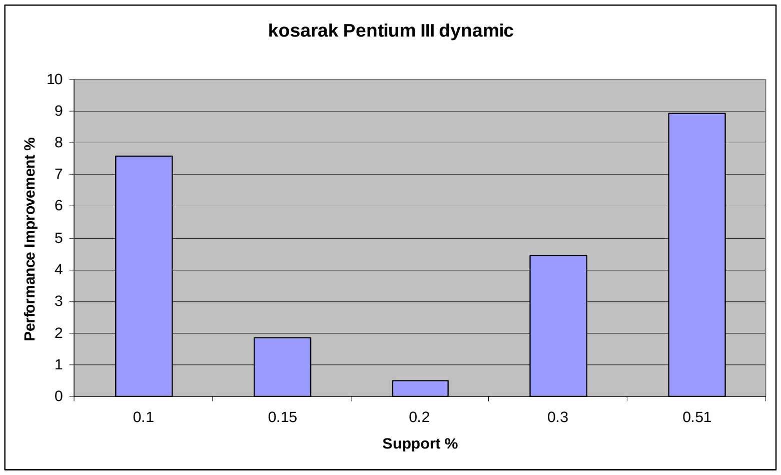


Figure 69: Performance Improvement of dynamic scheme for kosarak dataset on Pentium III

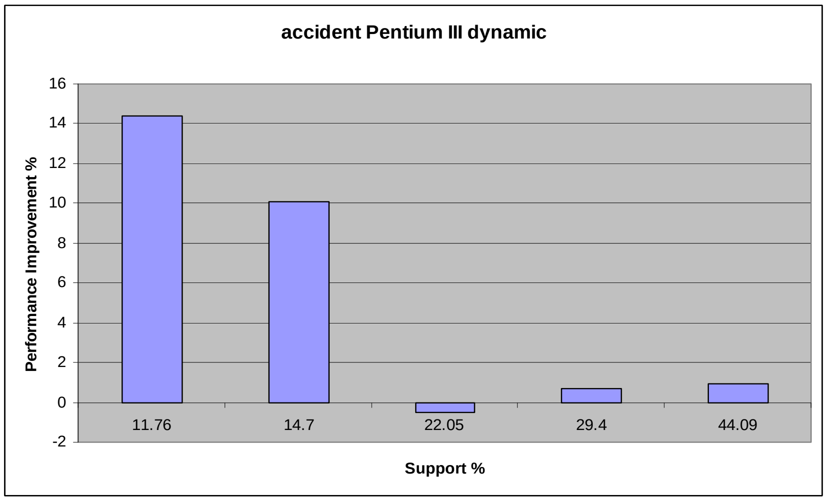


Figure 70: Performance Improvement of dynamic scheme for accident dataset on Pentium III

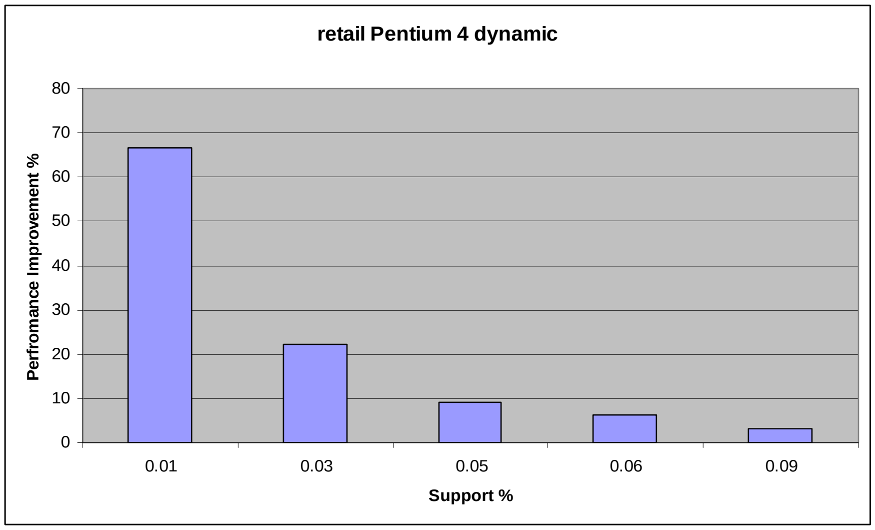


Figure 71: Performance Improvement of dynamic scheme for retail dataset on Pentium 4

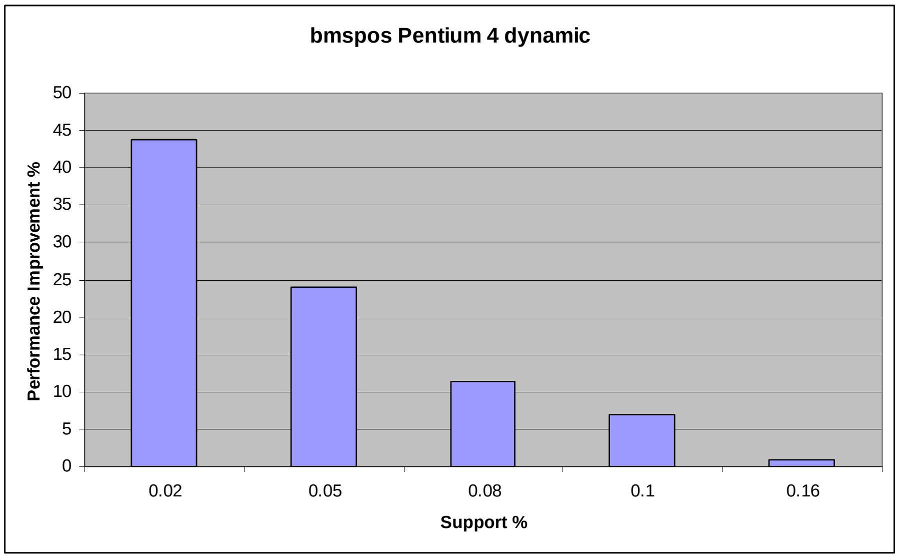


Figure 72: Performance Improvement of dynamic scheme for bmspos dataset on Pentium 4

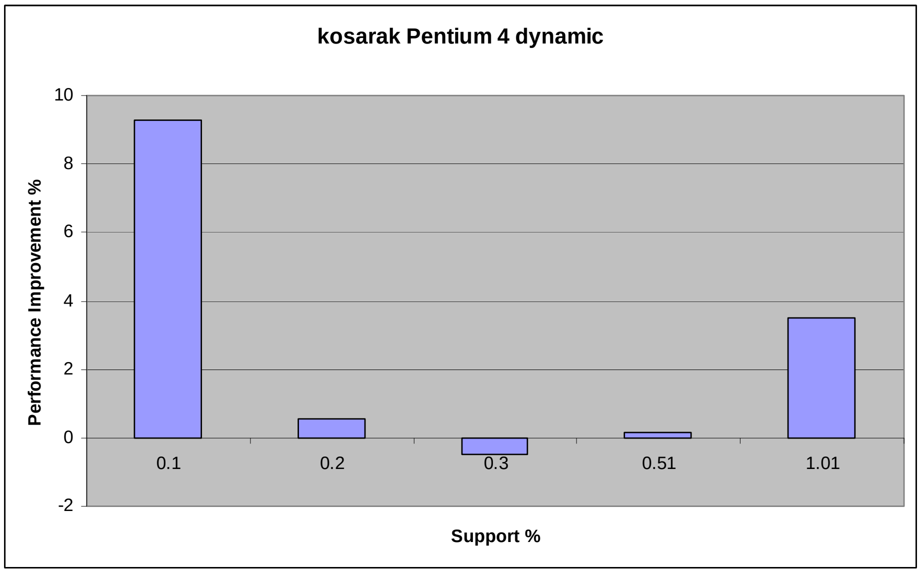


Figure 73: Performance Improvement of dynamic scheme for kosarak dataset on Pentium 4

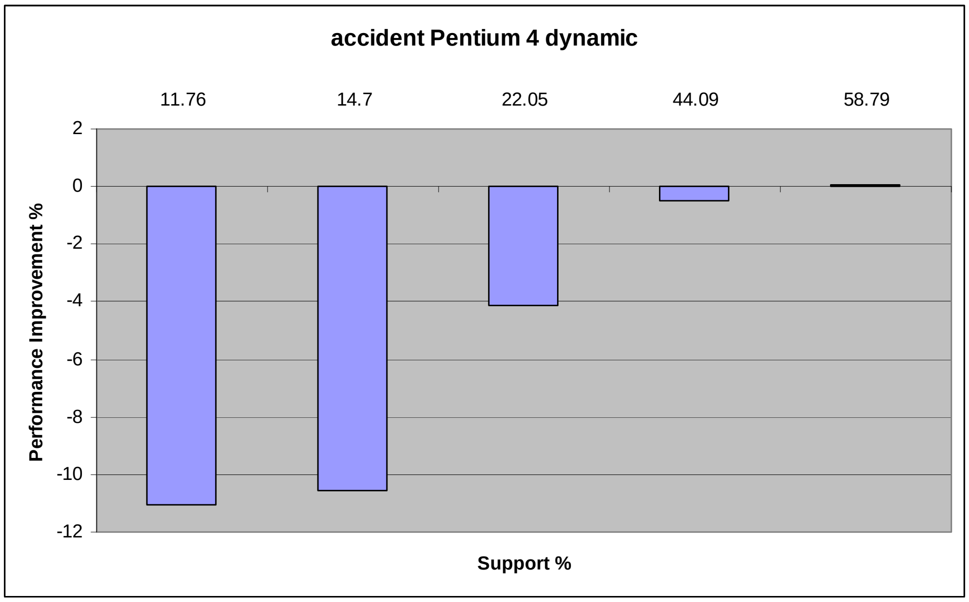


Figure 74: Performance Improvement of dynamic scheme for accident dataset on Pentium 4

Figures 75-78 compare the performance of different static sized node schemes and the dynamic scheme for a Pentium III system. Figures 79-82 present this comparison for Pentium 4 system.

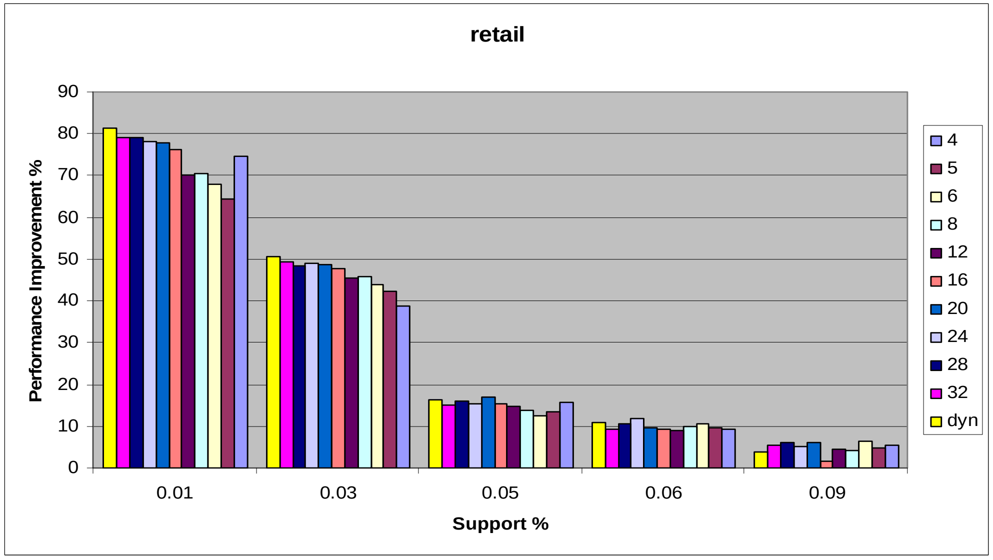


Figure 75: Performance Comparison for retail dataset on Pentium III

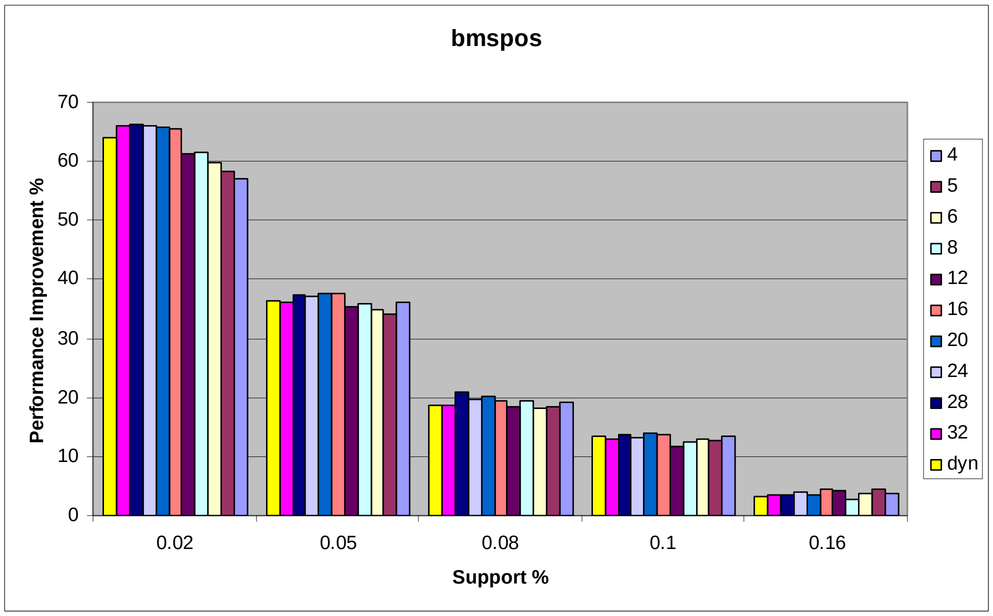


Figure 76: Performance Comparison for bmspos dataset on Pentium III

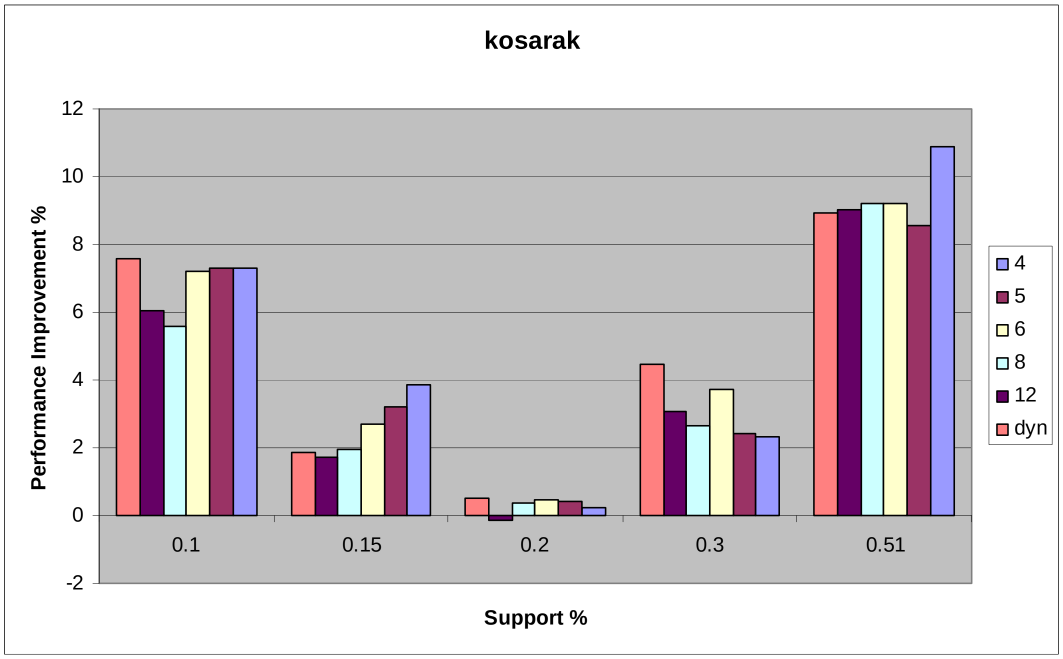


Figure 77: Performance Comparison for kosarak dataset on Pentium III

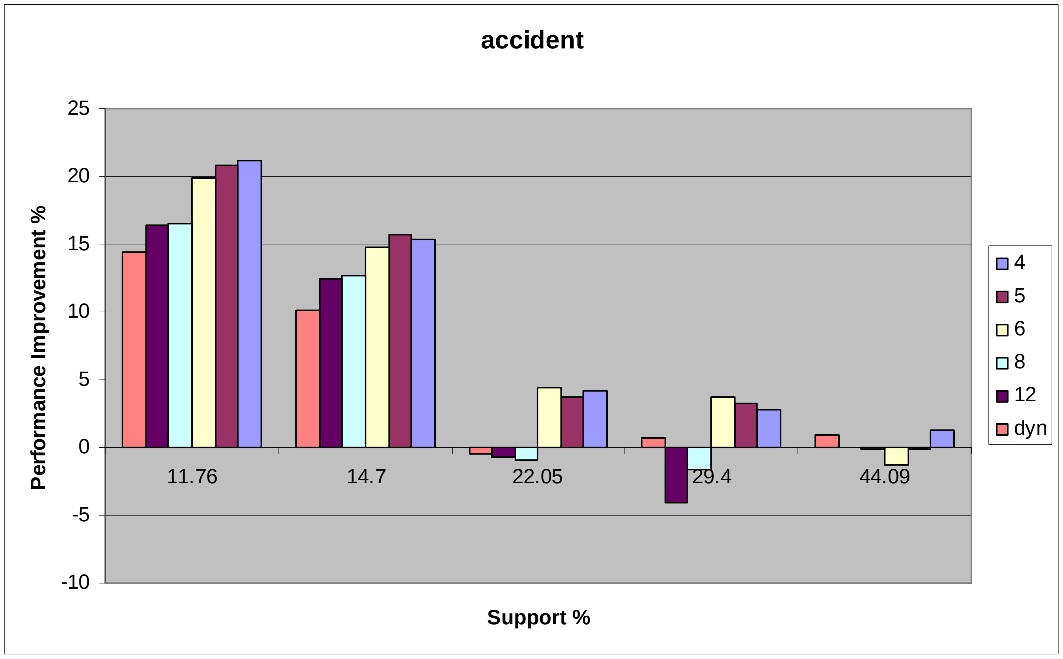


Figure 78: Performance Comparison for accident dataset on Pentium III

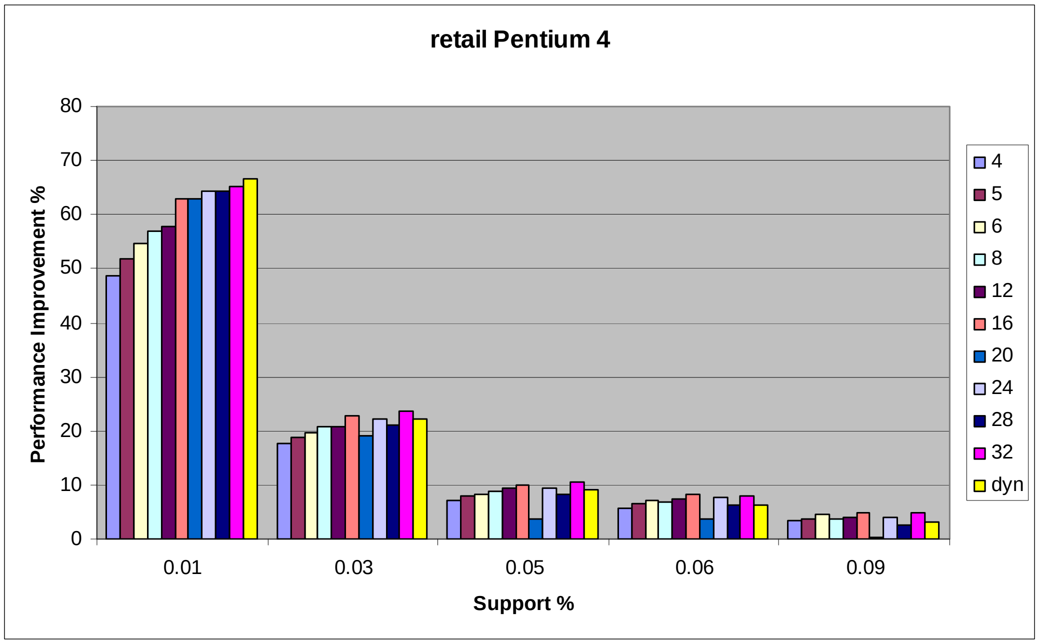


Figure 79: Performance Comparison for retail dataset on Pentium 4

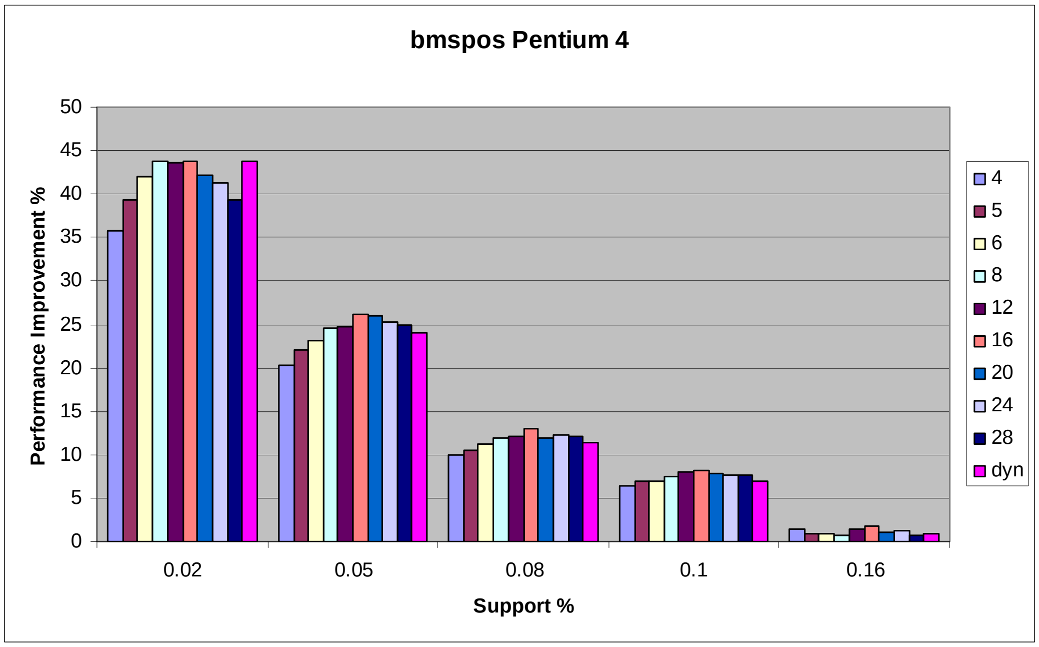


Figure 80: Performance Comparison for bmspos dataset on Pentium 4

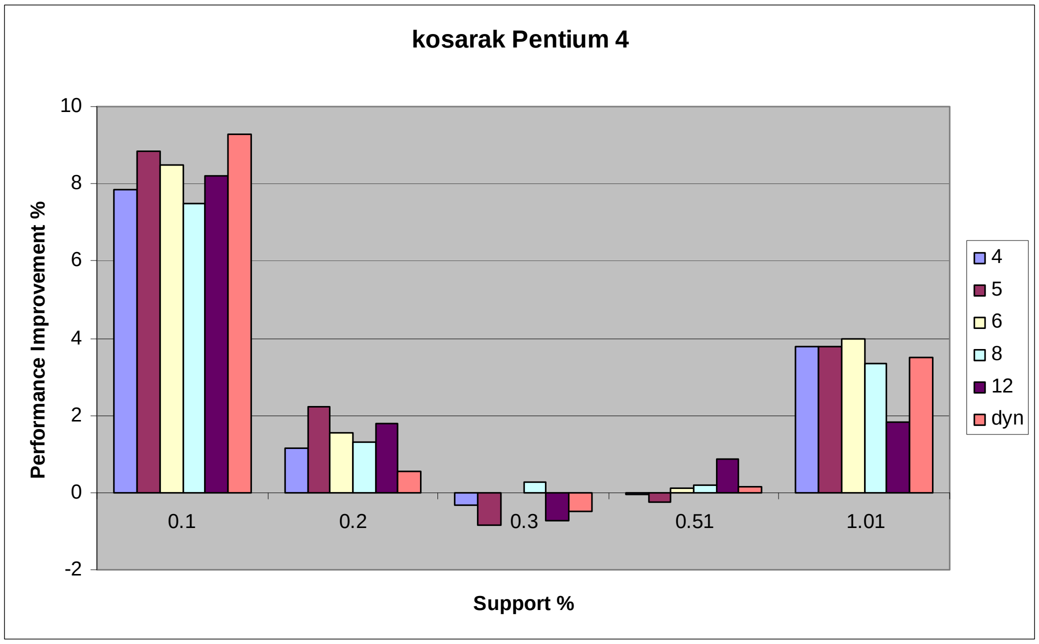


Figure 81: Performance Comparison for kosarak dataset on Pentium 4

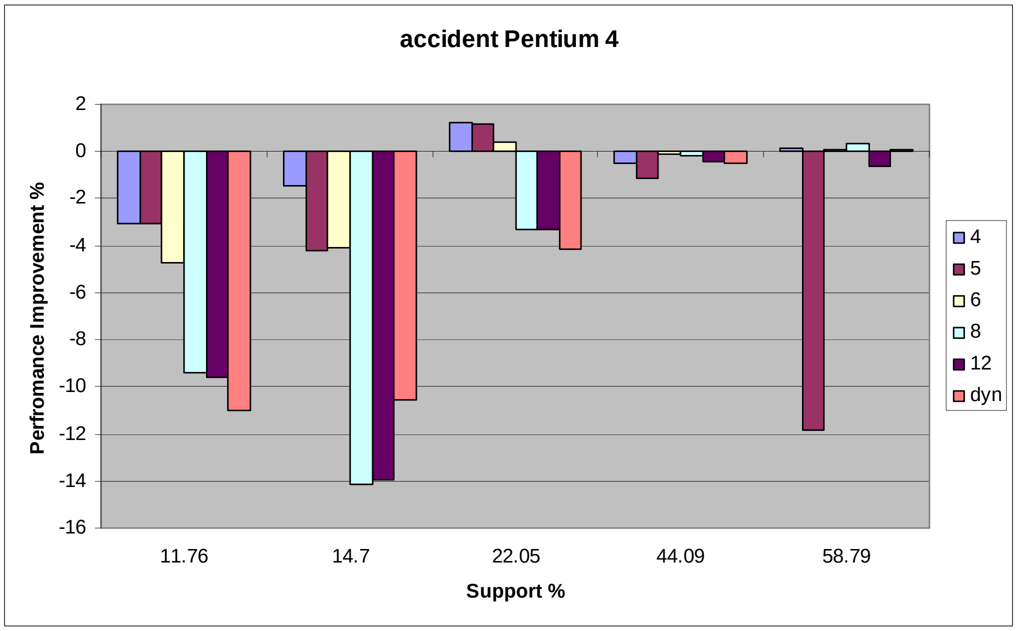


Figure 82: Performance Comparison for accident dataset on Pentium 4

From these figures we can see that the dynamic scheme performs consistently well across all datasets and support levels when compared with any single static scheme. Also the maximum difference in speedup of the best static sized node clustering scheme at each data point and the dynamic scheme was 6% on Pentium III and 8% on Pentium 4. More complex dynamic schemes can be tried to get further improvement.

3.1.10 Summary

We explored the memory hierarchy performance of a state-of-the-art CFI algorithm using real world datasets and a combination of tools such as OProfile, VTUNE, Simplescalar and iostat on both Pentium III and Pentium 4 systems. Profiling results show that cache performance becomes the bottleneck as the support level decreases. We used symbolic profiling to identify a bottleneck function that accounts for up to 69% of the L1 data cache misses, 86% of the L2 data cache misses, and 69% of the execution cycles. These cache misses occur due to traversal of the nodes of a prefix tree at a specific level through right sibling pointer dereferences. We then applied data structure modifications such as padding & aligning and static and dynamic node clustering to improve cache performance. Our results show that padding and aligning have no significant impact on performance. Static node clustering achieves performance gains of up to 79% on Pentium III systems and 65% on Pentium 4. Finally, we show that the dynamic allocation technique provides speedups up to 81% on Pentium III system and 67% on Pentium 4. The maximum difference in speedup of the best static sized node clustering scheme at each data point and the dynamic scheme was 6% on Pentium III and 8% on Pentium 4. This difference was for dense datasets, for sparse datasets dynamic scheme performed as well as or better than the best static sized node clustering scheme at each data point in a majority of the cases for both systems. The dynamic scheme we implemented is a simple one and there is room for getting further improvements using more complex schemes.

3.1.11 References

[1] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, September 1994.

[2] R. Agrawal and R. Srikant. Mining Sequential Patterns. In *Proceedings of the 1995 International Conference on Data Engineering*, pages 3–14, March 1995.

[3] T. Brijs, G. Swinnen, K Vanhoof, and G. Wets. The Use of Association Rules for Product Assortment Decisions: A Case Study. In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 254–260, August 1999.

[4] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Making Pointer-Based Data Structures Cache Conscious. *IEEE Computer*, 33(12):67–74, December 2000.

[5] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. Ullman, and C. Yang. Finding Interesting Associations without Support Pruning. In *Proceedings of the 2000 International Conference on Data Engineering*, pages 489–499, February 2000.

[6] K. Geurts, G. Wets, T. Brijs, and K. Vanhoof. Profiling High Frequency Accident Locations Using Association Rules, January 2003.

[7] Amol Ghoting, Gregory Buehrer, Srinivasan Parthasarathyand Daehyun Kim, Anthony Nguyen, Yen-Kuang Chen, and Pradeep Dubey. Cache-conscious Frequent Pattern Mining on a Modern Processor. In *Proceedings of the 31st International Conference on Very Large Data Bases*, page to appear, September 2005.

[8] G. Grahne and J. Zhu. Efficiently Using Prefix-Trees in Mining Frequent Itemsets. In *Proceedings of the 1st IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI)*, November 2003.

[9] J. Han, G. Dong, and Y. Yin. Efficient Mining of Partial Periodic Patterns in Time Series Database. In *Proceedings of the 1999 International Conference on Data Engineering*, pages 106–115, April 1999.

[10] Alvin R. Lebeck and David A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE Computer*, 27(10):15–26, October 1994.

[11] J. Levon et al. Oprofile.

[12] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of Frequent Episodes in Event Sequences. *Data Mining and Knowledge Discovery*, 1(3):259–28, March 1997.

[13] M. Martonosi, A. Gupta, and T. Anderson. MemSpy: Analyzing Memory System Bottlenecks in Programs. In *Proceedings of the 1992 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 1–12, June 1992.

[14] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering Frequent Closed Itemsets for Association Rules. In *Proceedings of the 7th International Conference on Database Theory*, pages 398–416, January 1999.

[15] M. Zaki. Generating Non-Redundant Association Rules. In Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 34–43, August 2000.

[16] Intel Inc. Intel VTune Performance Analyzers. http://www.intel.com/software/products/vtune/ .

[17] http://fimi.cs.helsinki.fi/

[18] Ron Kohavi, Carla Brodley, Brian Frasca, Llew Mason, and Zijian Zheng. KDD-Cup 2000 organizers' report: Peeling the onion. SIGKDD Explorations, 2(2):86-98, 2000.

[19] The Complexity of Mining Maximal Frequent Itemsets and Maximal Frequent Patterns. Guizhen Yang, Proc. of the 2004 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 344-353, 2004.

3.2 A Defect Tolerant Self-organizing Nanoscale SIMD Architecture

The continual decrease in transistor size (through either scaled CMOS or emerging nano-technologies) promises to usher in an era of tera to peta-scale integration. However, this decrease in size is also likely to increase defect densities, contributing to the exponentially increasing cost of top-down lithography. Bottom-up manufacturing techniques, like self-assembly, may provide a viable lower-cost alternative to top-down lithography, but may also be prone to higher defects. Therefore, regardless of fabrication methodology, defect tolerant architectures are necessary to exploit the full potential of future increased device densities.

This sub-section describes our defect tolerant Single Instruction Multiple Data, SIMD architecture. A key feature of our design is the ability of a large number of limited capability nodes with high defect rates (up to 30%) to self-organize into a set of SIMD processing elements. Despite node simplicity and high defect rates, we show that by supporting the familiar data parallel programming model the architecture can execute a variety of programs. The architecture efficiently exploits a large number of nodes and higher device densities to keep device switching speeds and power density low. On a medium sized system (~1cm^2 area), the performance of the proposed architecture on our data parallel programs matches or exceeds the performance of an aggressively scaled out-of-order processor (128-wide, 8k reorder buffer, perfect memory system). For larger systems (>1cm^2), the proposed architecture can match the performance of a chip multiprocessor with 16 aggressively scaled out-of-order cores.

Manufacturing defects, power density, process variability, transient faults, bulk silicon limits, rising test costs and multibillion dollar fabrication facilities are some of the challenges facing the continued scaling of CMOS. While architectural modifications (e.g., multicore) can provide some short-term relief, the semiconductor industry recognizes the importance of these issues and the need to explore long term alternatives to CMOS devices and fabrication techniques [17].

One promising alternative is DNA-based self-assembly of nanoscale components using inexpensive laboratory equipment to achieve tera to peta-scale integration. Although much of this technology is in its infancy (i.e., demonstrated in research lab experiments), by studying its potential uses for building computing systems, architects can gain a deeper understanding of its limitations and opportunities while providing important feedback to the scientists developing the new technologies.

DNA-based fabrication produces precise control within a small area (e.g., 9 µm^2) enabling the construction of a large number (~10^9-10^{12}) of small nodes (computational circuits with ~10^4 transistors) that can be linked together using self-assembly. This produces a random network of nodes, due to the lack of control over placement and orientation of nodes, which contain defective nodes and links. While our work is motivated by DNA-based self-assembly, it is applicable to any technology with similar characteristics (e.g., scaled CMOS with high process variability, high defect rates and point-to-point links between relatively small compute nodes). The challenge for computer architects is to efficiently exploit the computational power of the large number of nodes while overcoming two primary challenges: 1) loss of precise control over the entire fabrication process, and 2) high defect rates.

This sub-section presents a SIMD architecture designed to address these challenges. The fundamental building block in our architecture is a relatively small node (e.g., 1-bit Arithmetic Logic Unit, ALU with 32 bits of storage and communication support for four neighbors) that operates asynchronously. A configuration phase at startup isolates defective nodes and allows groups of nodes to self-organize into SIMD processing elements (PEs) which are connected in a logical ring, thus simplifying the programmer's view of the system.

Simulations using conservative estimates for node size and device speed show that the proposed design can match the performance of aggressively scaled architectures for 8 out of 9 benchmarks tested. Furthermore, this performance is achieved with a very low power density of 6.5 W/cm^2 (vs. >75 W/cm^2 for modern cores) while conservatively assuming that about 90% of the devices in the system switch every nanosecond. Finally, we show that our system can tolerate up to 30% defective nodes. Our results demonstrate the potential of this technology for building high performance architectures despite high defect rates and loss of precise control during fabrication. Further improvements are possible as the technology scales to allow more complex nodes, better inter-node connectivity, and faster devices. Our main contributions are:

1. Adapting self-organization methods to computer architectures.

2. Designing a node that balances fabrication constraints with functionality needed to communicate, compute, and self-organize.

3. Demonstrating the above capabilities by composing a high-performance, defect tolerant SIMD architecture from a random network of nodes.

3.2.1 DNA-based Self-Assembled Nanoscale Systems and the Architectural Implications

Self-assembly of nano-electronic devices has the potential to emerge as a lower cost alternative to top-down manufacturing. DNA-based self-assembly [32] uses the precise binding rules of DNA with nanoscale devices to build computing systems. We assume a proposed assembly process [26] to place electronic circuits on a DNA grid [38, 39]. The basic principle is to replicate a simple unit cell on a large scale to build a circuit. The unit cell consists of a transistor placed in the cavity of a DNA-lattice. A key requirement of this process is the ability to control the placement of electronic devices (e.g., carbon nanotubes [3, 9] or silicon nanowires [15]) at specific points on the DNA scaffold to form a circuit. Recently, two critical steps towards this goal were demonstrated: 1) aperiodic patterns, with a 20nm pitch, on a DNA grid [25] and 2) DNA-guided self-assembly of nanowire transistors [35]. We currently assume only two layers of metal interconnect within a lattice, which limits our ability to place and route circuits. We propose the use of conducting metallic planes separated by insulating layers to provide power and ground to the circuit. Figure 83 depicts a cross-sectional view of the lattice, with two layers of interconnect and the power and ground planes.

Current self-assembly processes produce limited size DNA grids and thus limit circuit size. However, the parallel nature of self-assembly enables constructing many nodes (~10^9-10^{12}) that may be linked together by self-assembled conducting nanowires [39]. The proposed self-assembly method does not control the placement and orientation of nodes as they are interconnected, resulting in a random network of nodes that contains defective nodes and links. Communication with external CMOS circuitry occurs through a metal junction ("via") that overlaps several nodes but interfaces with the network of nodes through a single "anchor node". There may be several via/anchor node pairs in large networks. Figure 83 shows a small network of nodes, including regions with defective links, and a via/anchor. In the rest of the sub-section we use the term "anchor" to refer to an anchor node/via pair.

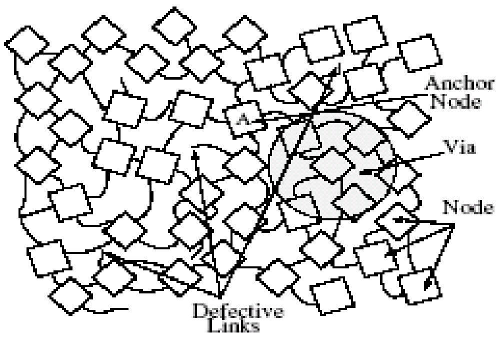


Figure 83. Self-assembled network of nodes.

A computing system built from this random network must: a) tolerate node and interconnect defects, b) not rely on underlying network structure, c) compose more powerful computational blocks from simple nodes, d) minimize communication overheads, and e) achieve performance that is at least comparable to future CMOS based systems. Several research projects examine building computing systems with a subset of these goals, including self-organization [1, 34], routing and resiliency in the face of defects [1, 16] and the ability to compose complex computational units from simpler blocks [23], but we face added challenges because of the extremely limited computational capabilities available in nodes. Our previous work, the nanoscale active network architecture (NANA) [29] is a general purpose architecture designed with a similar set of goals, assuming similar underlying technology. However, it fails to match the performance of conventional CMOS systems since it is unable to efficiently utilize the computational capabilities of the nodes at the same time. The design of the SIMD architecture presented in this sub-section is guided by the lessons learned through the design and evaluation of NANA.

3.2.2 System Overview

The goal of this work is to build a defect tolerant computing system with a random network of nodes using a mix of new solutions and adaptations of known techniques and achieve performance comparable to future CMOS based systems. To efficiently utilize large numbers ($>10^9$-10^{12}) of nodes we implement a SIMD architecture and focus on data parallel workloads. Our proposed system - called the "Self-Organizing SIMD Architecture" (SOSA) - supports a three operand register-based Instruction Set Architecture, ISA with predicated execution and explicit Processing Element-Shift (PE-Shift) instructions to move data between PEs and communicate with an external controller. We assume that the external controller has access to a conventional memory system.

Each self-assembled node is a fully asynchronous circuit and there is no global clock to synchronize data transfers between or within nodes. Each node has a 1-bit ALU with a small register file and connects to other nodes with (up to four) single wire links. Each link supports low bandwidth asynchronous communication that transfers 1 data bit per handshake. To support deadlock-free routing, we add support for three virtual channels (1 bit each). The random network of nodes is organized at two levels during a configuration phase. First, since a node is too small to hold a PE, we group sets of nodes to form a PE. Second, PEs are linked in a logical ring providing programmers a simplified system view to reason about inter-PE communication.

The configuration process, initiated from an anchor, maps out defective nodes and connects functional nodes in a broadcast tree. The system can be configured in two ways: a) as a monolithic system, all nodes on one logical ring (one "cell"), or b) as multiple, independent logical rings (multiple "cells"). For a monolithic system, anchors can be used to speed up PE configuration and data input/output by serving as "taps" into the logical ring. The only constraint enforced during configuration is that an anchor cannot partition a PE. In case (b), we achieve space partitioning by running the configuration algorithm from multiple anchors to create independent cells. Space partitioning is a common technique used in highly parallel systems to increase resource utilization by enabling the execution of multiple instances of one workload, or running multiple workloads.

3.2.3 Node Microarchitecture

Careful node design is critical in maximizing system performance. Due to limited node size, designing the node architecture involves a trade-off between maximizing functionality (compute, communicate, and self-organize) and performance while minimizing circuit size. To avoid the area and power overhead of routing clock signals and to mitigate the effects of device parameter variation, instruction execution and sequencing within a node are asynchronous.

3.2.3.1 Data Path

Each node has a simple data path that consists of a 1-bit ALU, a 32-bit register file, and a data buffer that stores incoming and outgoing data. The register file and data buffer can act as sources and/or sinks for the ALU. The data buffer cannot be written to unless the current instruction is waiting for data, and once written, cannot be overwritten until the data is used by the ALU. All internal node communication occurs on dedicated point to point links. Where possible, we overlap the latency of moving a bit between two parts of the node with other operations.

Nodes can be designed to partition the 32-bit register file into N-bit wide registers that require an N-bit ALU or repeated use of a single-bit ALU. For example, a 32-bit PE could be created with 32 1-bit registers, requiring 32 nodes for the PE, or with 16 2-bit registers, requiring 16 nodes to form the PE. Increasing register width increases the work done per instruction in a node, reduces the number of nodes required to form a PE, and reduces inter-PE communication overheads (since PE length reduces). However, for a fixed sized node, wider registers reduce the number of registers available to a programmer. Simulations reveal that 2-bit wide registers achieve the best trade-off in terms of maximizing the benefit of wider registers and the number of registers available to programmers. We also find that program performance is not sensitive to ALU execution latencies shorter than the time taken to send/receive a bit between nodes.

3.2.3.2 Control

The control logic in the node can be divided into two parts. The first part (configuration logic) is used only during configuration and has control registers for defect testing/isolation (main control register), and PE configuration (PE control register). Figure 84 shows a floorplan of the node with the configuration logic demarcated by a dashed rectangle within the control and data block.

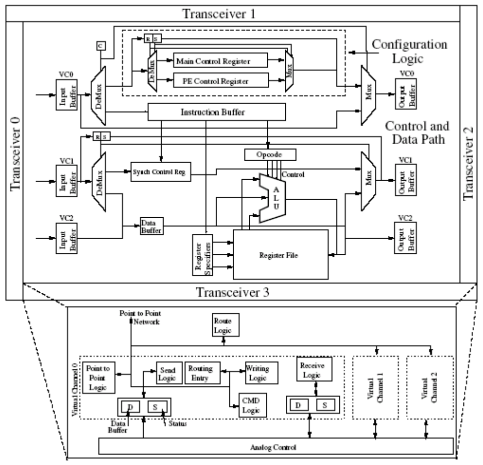


Figure 84. Node floorplan.

The second part is the run-time control logic used to decode and execute instructions. To reduce design complexity we sacrifice latency and use microcoded control logic with each instruction divided into multiple microinstructions. The run-time control logic has three control registers to hold each of three micro-instructions that comprise an instruction: a) opcode, b) register specifier and c) synchronization (synch). The synch microinstruction holds an optional counter value (“repeat counter”) to enable the repeated execution of one instruction and avoid broadcasting the same instruction consecutively. The register specifier includes fields that allow simple increment/decrement operations on register specifiers in conjunction with their reuse (for striding through registers). We add a shared circuit that is used to increment/decrement register specifiers and the repeat counter. Because of high instruction execution latencies, the increment/decrement operations can be overlapped with other operations, effectively hiding their latency.

All arriving microinstructions are first sent to an instruction buffer before they are moved to the control registers, creating a simple two-stage pipeline (buffer, execute). Each entry in the instruction buffer can hold all three micro-instructions that form a full instruction. The instruction opcode is fully decoded and copying the instruction into the control registers enables all control signals required to execute the instruction and detect its completion so that the next instruction can begin to execute. Increasing the instruction buffer size can improve performance by overlapping instruction broadcast with execution, but can also cause greater contention (and reduce performance) on the network since instructions and data must share link bandwidth. Simulations reveal that a single entry instruction buffer offers the best trade-off between improving performance and minimizing design complexity.

3.2.3.3 Inter-Node Communication

Nodes communicate with each other on single-bit asynchronous links. Each end of a link terminates in a transceiver that can handle three virtual channels (using 1-bit buffers per virtual channel). The transceiver can route each virtual channel (VC) independently and requires three bits of state per VC to store the destination address. To support self-organization, nodes include logic to configure static routes. Virtual channel 0 (VC0) is used to broadcast instructions. Virtual channel 1 (VC1) and virtual channel 2 (VC2) are used to route data in opposite directions on the logical ring. Each asynchronous transaction on a link is controlled through a four-phase handshake. The links support bidirectional full-duplex transfers. To simplify transceiver circuit size and complexity we transfer 1 bit per handshake (which severely limits link bandwidth).

3.2.3.4 Circuit Size and Power Estimates

We have completed the circuit design for all node components. We use this design in conjunction with layouts of simple logic blocks to estimate node size and power consumption. Our simulator models the system in sufficient detail to make it relatively easy to extract a circuit model for most components. Figure 84 shows a floorplan of a node, showing the approximate position (not to scale) of the datapath, control and transceivers. We estimate that the entire node will require 10,000 transistors. Since the proposed fabrication technology currently imposes limitations on the number of metal layers, we estimate the final area of the node to be the equivalent of 22,000 transistors (based on our experience in laying out circuits) which translates to a 3µm x 3µm node. Recent work [39] has shown that it should be possible to manufacture DNA grids of this size.

To estimate system power consumption, we use the energy xdelay product for carbon nanotube field effect transistor (CNFET) circuits [11]. Based on a switching speed of 1 ns, and estimated node gate and latch counts, we calculate an upper bound on the per node power consumption. During execution, the configuration logic and a large part of the register file are inactive (at most 3 registers can be active). Accounting for these inactive elements yields a node activity factor of 0.88, which corresponds to a power consumption of 0.775µW per node. To obtain an upper bound on the power density of this system, we assume that nodes are packed with no space between them. Using our estimated node area (9µm^2) and power (0.775µW), we get a maximum power density of 6.5W/cm^2, with a node activity factor of 0.88. This is much less than the power densities of current processors, which are greater than 75 W/cm^2. This estimate is pessimistic since the activity factor is a conservative estimate, we cannot pack nodes perfectly, and defective nodes will further reduce power density.

3.2.3.5 Summary

Each node in SOSA is a small circuit that can communicate with up to four neighbors, store small amounts of state and perform simple computation. To minimize area and power overheads the nodes use asynchronous logic, however like current processors we still dedicate significant area to control and communication circuitry. The challenge is to coordinate the operation of these nodes connected through an unstructured network to execute programs.

3.2.4 System Configuration

To use the random network of nodes to perform useful computation we use a configuration mechanism to impose logical structure on the network and isolate defective nodes and links from the rest of the system. This allows nodes to self-organize and avoids the need for an external defect map, which would be impractical to obtain given the scale and bandwidth limitations of the system. Once defective nodes are isolated, the functional nodes are grouped to form PEs. We now describe this configuration in detail.

3.2.4.1 Logical Structure and Defect Isolation

We use a variant of the “reverse path forwarding” (RPF) algorithm [7,27] to impose a logical tree structure on the network and isolate defects. When the system is powered up or reset, all nodes enter a “configuration mode”, steer incoming packets to the configuration control registers and execute the distributed RPF algorithm. A small packet is inserted through an anchor and is broadcast on all of its active links (the transceiver analog control circuitry tests the liveness of its physical link).

The RPF algorithm states that any node receiving the broadcast propagates it on all links except the receiving link if and only if the node has not seen the broadcast before. The node also stores the direction (“gradient”) from which it received the broadcast and sets up internal routing information based on this direction. Following the gradient through a set of nodes leads to the broadcast source—the tree root. A depth first traversal is established by nodes locally selecting links in a predefined order relative to their gradient link. Opposite orderings are used for forward (VC1) and reverse (VC2) traversals. This method can be used to have all nodes in the system self-organize into a tree or it can be used to create multiple trees by initiating the broadcast through multiple anchors. For example, we could self-assemble the random network of nodes on a silicon wafer with a grid of vias to create a system with multiple anchors.

Defect isolation is achieved by 1) augmenting each node with built-in-self-test and assuming fail-stop behavior [28], and 2) including a simple test vector in each broadcast packet that each node must successfully execute before propagating the broadcast. Nodes failing the test are isolated since there is no path through the node. Simulations show that the gradient can reach a very large fraction of functional nodes (i.e., achieve good coverage) for node defect rates up to 30%. Handling more complex defects like Byzantine failures is beyond the scope of this work.

3.2.4.2 Configuring Processing Elements

A node is too small to hold an entire PE, so we logically group a set of nodes to form a PE. To create PEs with N bits (we assume N=32), we traverse the broadcast tree in depth-first order (on VC1) and group N+2 consecutive unconfigured nodes. We use one configuration packet per PE. An unconfigured node receiving a configuration packet examines it to determine what node in the PE is to be configured next. The first node holds auxiliary control bits for the PE and is called the “head” node. The next N nodes serve as compute nodes that form the N-bit PE. The last node (“tail”) serves as the terminating point of the PE and is used to store the status bits (carry/borrow) resulting from an arithmetic operation. A newly configured tail node sinks the configuration packet. To minimize PE setup time in large networks ($>10^9$ nodes), we could distribute configuration by exploiting multiple anchors.

If the broadcast tree does not have sufficient nodes to form an integral number of PEs, the “incomplete” PE is deconfigured before execution begins by performing a reverse depth first traversal on VC2. PE deconfiguration uses a simple packet and starts with the last configured node of the partial PE (i.e., PEs with no tail), and deconfigures all intermediate nodes until it reaches (and terminates at) the head node.

We extend PE configuration to optimize PE length (hops from head to tail). Very long PEs (e.g., a PE that spans the broadcast tree root) may reduce performance due to longer intra-PE communication latencies. Since the post-configuration step deconfigures partial PEs, a PE that crosses a length threshold can be rejected by starting a new PE without creating a tail node. We empirically find that a threshold of 4 times the minimum PE length (compute nodes + head + tail) achieves a good balance between extra nodes required and performance gained by reducing PE length.

Once PEs are configured, all nodes set a “run” mode bit. Packets are no longer routed to the configuration control registers, unless the node receives a global reset instruction. Each PE waits for instructions to execute. In the next section, we describe how SOSA uses the configured PEs to execute instructions.

3.2.5 System Architecture

In this section, we describe the architecture of SOSA. Careful node design coupled with the self-organizing capability of each node enables us to map a data parallel architecture onto the random network of nodes. We begin by describing the instruction set and execution model. Then, we present an example illustrating the execution of an instruction in the system.

3.2.5.1 Instruction Set Architecture

SOSA uses a three register operand ISA, with microcoded instructions (Table 8 shows a subset of the instruction set). A full instruction has between 39 and 44 bits and contains: a) a 16-bit fully-decoded opcode microinstruction, b) a 20-bit register specifier microinstruction (4 bits per register specifier for a 16-entry register file, and 2 extra bits per register specifier to allow increment/decrement/no change operations), and c) a 3-bit “synch” microinstruction with an optional 5-bit synch repeat counter. Each microinstruction can be independently broadcast and includes 2 bits of control overhead to select a control register as a destination.Since opcodes are fully decoded, it is relatively straightforward to support fused instructions that include combinations of operations to increase the work done per instruction. For example, a Copy-Shift first copies the source to the destination register, and then performs a shift operation on the destination register. SOSA also supports predicated instruction execution (all instructions can be predicated) and has three types of instructions that can modify predicate bits: a) conditional instructions, b) unconditional predicate modifying instructions and c) predicate-shift instructions.

Table 8. Instruction set architecture.

Instruction Type	Opcodes	Description
Arithmetic	ADD, SUB, INC, DEC, SETGT, SETLT, SETEQ, SETNEQ	Various arithmetic and conditional instructions, "Set" instructions set the specified predicate register if the condition is satisfied
Logical	AND, XOR, OR, NOT	Various logical instructions
Shift	SHIFTML,SHIFTLM, PSHIFTML	Various SHIFT instructions. ML=>MSB to LSB, LM=>LSB to MSB. The prefix "p" indicates that the instruction modifies the specified predicate register (not a predicated instruction)
PE-Shift	SHIFTMLPE, SHIFTLMPE	PE-Shift instructions. Move register to adjacent PE
Register operations	CLEAR, CPREG, SWAP	Clear, Copy or Swap registers
Predicated	PR[OPCODE]	Any instruction with the prefix "Pr" is predicated. The predicate register corresponds to the first source register
Fused	CPSHIFTLM, CPSHIFTML	Copies source into destination, and performs a shift on the destination
Signal	SIG_CTRL	Send signal to external controller

Data exchange with the external controller and between PEs is handled through PE-Shift instructions. When PEs in a cell execute a PE-Shift instruction, each PE sends the contents of the specified register to a neighbor (left or right), and receives a new value for the register from the other neighbor (right or left). Since these instructions are critical for data communication, it is important to minimize their latency. We optimize PE-Shifts using the following observation: for a N-bit PE, each bit moves exactly (N+2) positions to the left or right, and a node only needs to store the (N+2)th bit in its register file and can "forward" the remaining bits without register access. We use the synch repeat counter to track the bits being forwarded by the node. The node stops forwarding when it receives the (N+2)th bit. When a node is "forwarding" data, it copies the data bit directly from its input buffer to its output buffer. This reduces the critical path of a bit through the node.

3.2.5.2 Execution Model

Instructions are broadcast on VC0 to all nodes, thus PEs, in a cell. Nodes first place instructions in the instruction buffer and then forward them down the broadcast tree. Instruction broadcast stalls when the instruction buffer is full. The arrival of the synchronization micro-instruction is a signal to the node that all parts of the instruction have been received. An instruction moves from the instruction buffer to the node's internal control registers only when the previous instruction finishes execution. Since nodes are bandwidth limited, we allow the partial broadcast of instructions to reduce the number of bits broadcast. If an instruction broadcast skips a microinstruction (except synch), we reuse the previously latched value from the corresponding control register. The synch repeat counter also helps reduce the number of bits broadcast.

Non-predicated instructions can be executed independently by nodes of a PE, if there are no inter-bit data dependencies (e.g., for an OR instruction). The head and tail nodes act as PE delimiters, and ensure that intra-PE data packets do not cross PE boundaries. The tail node also stores the carry/borrow out from arithmetic operations. The head node stores predicate bits (one per physical register) that are used to conditionally execute predicated instructions. The head node reads the specified predicate bit and informs the remaining nodes in the PE whether the predicated instruction is to be executed or squashed by sending a synch microinstruction on VC1. Since each node in a PE must wait for the extra synchronization microinstruction (which is consumed by the tail), execution of predicated instructions is serialized through a PE.

3.2.6 Evaluation

This section describes our evaluation methodology, simulation infrastructure and workloads, then compares SOSA performance to four other architectures. We find that SOSA achieves good performance on benchmarks that have data parallelism. For a configuration with more than 64K PEs, SOSA matches the performance of an ideal 16-way Chip-level MultiProcessor, CMP. Thus, despite SOSA's severe limits on node computational power, network bandwidth and connectivity, and low control over the fabrication process, it matches the performance of idealized conventional architectures, with lower device switching speeds and a lower power density. We then show that SOSA can tolerate high node defect rates. For the encryption benchmarks, performance gracefully degrades as the fraction of defective nodes increases to 30%. For the other benchmarks, by over-provisioning the system, SOSA tolerates up to 20% defective nodes with a small (<10%) degradation in performance. We also find that the instruction buffer and microinstruction reuse optimizations improve performance. Increasing ALU execution latency does not impact performance so long as it is lower than communication latencies.

3.2.6.1 Methodology

We evaluate SOSA using a custom, event-driven simulator and use results from simulating smaller systems to extrapolate the behavior of larger systems. Since the nodes do not use a clock, we define the time taken to perform one part of the inter-node asynchronous communication handshake as one "time quantum". The latency of all activity in the node is a multiple of this time quantum. Experimental devices are expected to operate at frequencies exceeding 100 GHz [4] with demonstrated frequencies over 10GHz [33] (time quantum of 0.1 ns), and asynchronous handshakes at high speeds have been demonstrated for high bandwidth crossbar networks [21]. We expect SOSA's performance to scale with device performance, but assume a conservative time quantum of 1 nanosecond to avoid over-estimating performance due to aggressive technological parameters. We list our default simulation parameters in Table 9. We use a custom tool that models the growth of DNA nanotubes between nodes to generate network topologies.

Table 9. SOSA system parameters.

Parameter	Value	Parameter	Value	Parameter	Value
Register File	16 entry, 2-bits per node	Synch Repeat Counter Width	5 bits	Data Width	32 bits
Time Quantum	1 ns	PE Length Optimization	Enabled	Instruction Buffer Size	1 entry
ALU Latency	1 time quantum	Register Increment/Decrement	Enabled	Link Type	Full Duplex

We compare the performance of SOSA to a Pentium 4 (P4) (3 GHz, 1MB L2, 1 GB RAM), an ideal out-of-order superscalar (I-SS) (128-wide, 8k ROB, 1-cycle memory latency), an ideal 16-way CMP (16-CMP) (obtained by linearly scaling performance of the I-SS) and an ideal implementation of SOSA (I-SOSA) that uses the same instruction set, but assumes unit instruction execution latencies, and no communication overhead. Table 10 lists the parameters used to simulate the I-SS with SimpleScalar [2].

Table 10. Ideal superscalar parameters.

Parameter	Value	Parameter	Value	Parameter	Value
Width	128 (Fetch/Decode/Issue/Commit)	Integer ALU	128 Add,128 Mul	Branch Prediction	Perfect
Instruction Fetch Queue	1024 Entries	FP ALU	128 Add,128 Mul	Memory Latency	1 cycle
ROB/LSQ	8192 entries, single cycle access	Frequency	10GHz	Memory Ports	128

Table 11 contains brief descriptions of the test programs, the broad application classes they fall under, and the number of PEs required by SOSA to run one instance of a program. For all programs other than the encryption algorithms, we configure the system as a single cell with the necessary PEs. For the encryption algorithms, we configure the system as a collection of cells, each of which operates as a pipelined encryption unit. We use GNU Complier Collection, gcc to generate PISA binaries for simple scalar (flags: -O3) and Intel's C Compiler (icc, flags: -O3 -fast -tpp7) for the P4 since optimized icc binaries outperform optimized gcc binaries. We test several versions of matrix multiplication from [31] and identify the best version for the P4 (naïve version with three nested loops, since icc vectorizes loops for the Streaming SIMD Extension (SSE) units) and I-SS (static loop unrolling). For sorting, we use an implementation of quicksort. For SOSA each program is hand-optimized (e.g., loop unrolling, code re-organization). The SOSA code for matrix multiplication and the image filters assumes data is in place before execution begins. However, this overhead forms only a small fraction of total execution time and can be reduced by exploiting multiple anchors in the system. The other workloads explicitly account for I/O overheads. The running times of programs do not include system configuration time (which is proportional to the number of nodes in the system). To estimate SOSA performance for configurations with more than 16K PEs, we use simple linear extrapolation (simulating a 256x256 matrix multiplication on a 3 GHz P4 with 32 GB RAM takes ~50 days, which is impractical for data collection purposes). To validate the extrapolations we compare extrapolated run times to simulated run times for large configurations (8K-16K PEs).

Table 11. Benchmark descriptions

Application Class	Benchmark Description
Scientific	*Multiply* integer *NxN matrices* (N^2 PEs)
Image Processing (Filters)	Apply a *generic 3x3 filter* on an NxN image (N^2 PEs)
	Apply a separable *gaussian filter* on an NxN image (N^2 PEs)
	Apply a *median filter* to an NxN image to reduce noise (N^2 PEs)
General Purpose	Odd-Even Transposition *Sort* [19]- Parallel sort with nearest neighbor communication (N PEs for sorting N numbers)
Cryptography	Tiny Encryption Algorithm *(TEA)* - Simple encryption algorithm used in the XBox (64 PEs)
	eXtended TEA (*XTEA*) - Eliminates known vulnerabilities in TEA (64 PEs)
Search	*Search* a database for a match with an input 32 bit string (O(N) PEs for N strings)
Bin-Packing	Pipelined version of *bin-packing* with first-fit heuristic (N PEs for N bins)

3.2.6.2 Results

We now examine the performance of applications on SOSA with no defects. SOSA provides users the flexibility to configure the system to minimize program running time (single cell, single program instance), or to maximize throughput (multiple cells, one program instance each). We divide our evaluation in two parts based on the performance metric being used (execution time or throughput).

Execution Time. For many workloads (image filters, matrix multiplication, sorting), system performance is determined by program execution time since we are solving a single instance of each problem. To evaluate the performance of these programs on SOSA, we configure the system to create one cell with the required number of PEs. The latency of an individual instruction in SOSA is high due to the overheads caused by limited node capabilities. However, SOSA can amortize this overhead by executing the same instruction in all PEs at the same time. Hence, we expect SOSA to perform poorly for small input sizes, where each instruction is executed in a small number of PEs. However, SOSA performance should improve as input size increases and eventually match (or exceed) the performance of the P4, I-SS and 16-CMP. The input size at which SOSA outperforms a particular architecture is application dependent.

Inspecting the main loop body for matrix multiplication in Figure 85 (optimizations are omitted to keep the code compact and readable), we see that the primary advantage for SOSA is the simultaneous computation of all products in the N^2 PEs. This allows SOSA to convert the $O(N^3)$ algorithm to $O(N^2)$. Image filters and sorting are reduced from $O(N^2)$ algorithms to O(N).

```
  ; Initialize before Multiply
CPREG R4,R2      ;Copy R4->R2
CPREG R3,R1      ;Copy R3->R1
CLEAR R5         ;Clear R5
  ;Multiply (Loop Dw times) (Dw: Data Width)
SHIFTLM R1       ;Shift LSB to MSB (multiply by 2)
PSHIFTML R2,R5   ;Shift MSB->LSB, LSB->pred.reg R5
PRADD R5,R1,R5   ;if predicate is set, R5=R5+R1
CLEAR R6          ; Clear R6
  ; Accumulate partial products
  ;Repeat log2(N) times (i is iteration count)
ADD R6,R6,R5     ;Accumulate partial sum
CPREG R6,R5      ;Copy R6 to R5
SHIFTMLPE R5     ;Repeat i*2 times
  ; End Repeat
ADD R6,R6,R5     ;Final add
  ; Align rows of matrix A for next set
  ; of multiplies (Repeat N times)
SHIFTMLPE R4     ;Move A 'N' PEs to the left
  ; Move Result
CPREG R8,R9      ;if R8==1, this PE holds the first
                 ;row/column element, move to R9
PSHIFTML R9,R6   ;Move that bit into the predicate
                 ;register R6
PRCPREG R6,R7    ;if predicate set, copy R6->R7
SHIFTMLPE R7     ;Move R7 one PE to the left
```

Figure 85: Matrix multiply: assembly code w/o loop unrolling.

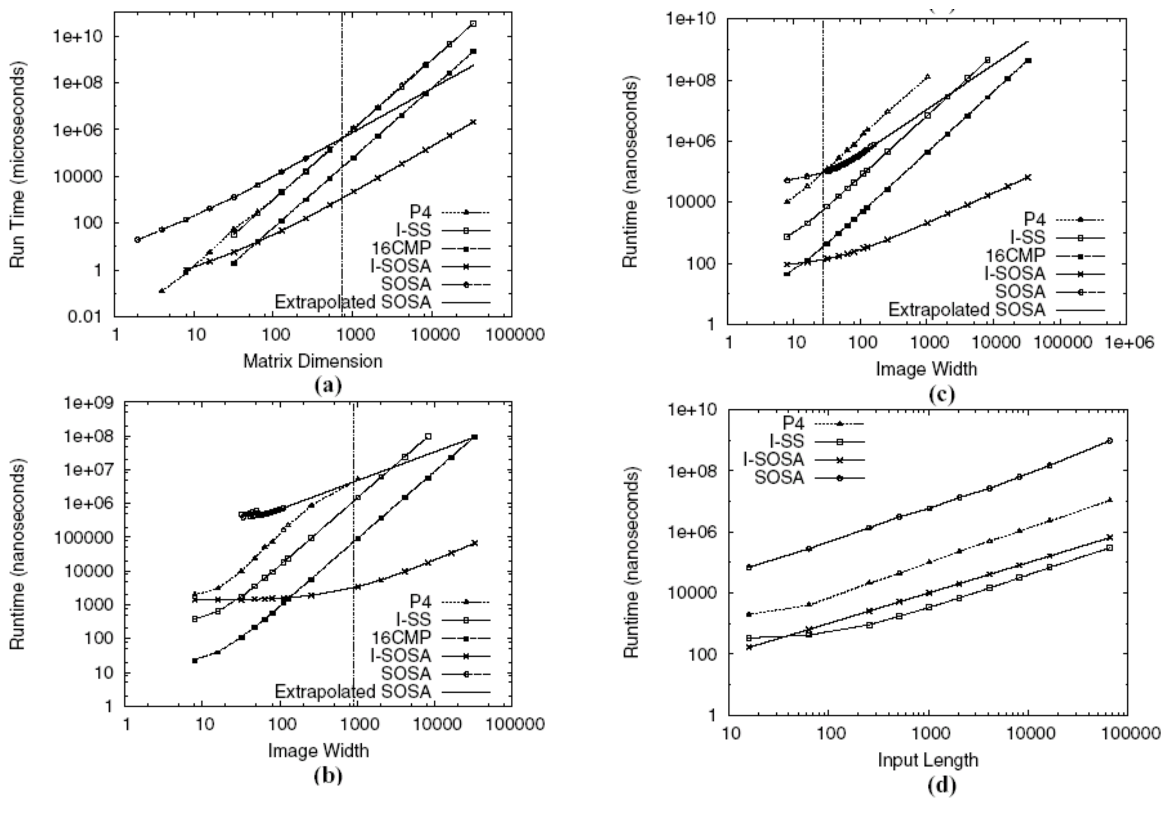


Figure 86: Single Cell Program Runtimes: (a) Matrix Multiplication, (b) Gaussian Filter, (c) Median Filter and (d) Sort. The vertical line denotes the input size beyond which SOSA does better than the Pentium 4.

We plot the running time of matrix multiplication, Gaussian filters, median filters and sorting on different architectures in Figure 86, marking the input size beyond which SOSA outperforms the P4 with a vertical line (results for the generic 3x3 filter are qualitatively similar to the Gaussian filter, and are skipped due to space constraints). As expected, SOSA does worse than the conventional architectures for small input sizes, but matches and overtakes them as input size increases (except for median filter and sort). The P4 matches the I-SS on matrix multiplication for two reasons: a) the P4 makes use of its SSE units, and b) I-SS only achieves an IPC of 9. The P4 performs much worse without the SSE units.

The performance of the median filter and sort algorithms is limited by their dependence on predicated instructions which serialize execution in a PE. While the number of predicated instructions in the median filter is fixed (independent of input size), for sort it scales with input size. For the median filter, SOSA is able to match the performance of the uniprocessors, but not the ideal 16-CMP (for image sizes up to 16Kx16K). For sort, the potential speedup on SOSA over quicksort on a single processor (average case) is O(log(N)). However, the overhead introduced by predicated instructions makes it impossible for SOSA to match the performance of the I-SS or P4. Exploring techniques to reduce this overhead is future work. Note that even I-SOSA cannot outperform the I-SS at sorting. This highlights one key limitation of SOSA: it is not a general purpose architecture and cannot match the performance of conventional processors on general purpose workloads.

Throughput. There are a large number of workloads where high system throughput is desirable. The parallel computational capabilities of SOSA can be used to achieve high system throughput by dividing the system into multiple cells, each having a set of PEs. While there are multiple ways to improve throughput, we focus on using multiple instances of a single application (operating on different data) running on different cells. For example, if we assume an area of 100mm^2 (approximately the area of a P4 in 90nm CMOS), we can configure over 5,000 cells (assuming an average inter-node gap of 1µm) that each perform an 8x8 matrix multiplication and achieve much higher throughput than the P4 or the I-SS.

TEA [37] and XTEA [24] are two simple encryption algorithms developed at the University of Cambridge that use a combination of shift, add and xor operations to encrypt 64 bit blocks of data with a 128-bit key, with XTEA requiring more operations per iteration to achieve better cryptographic security. We implement pipelined versions of both algorithms that require 64 PEs (corresponding to 64 encryption iterations) in a cell. Due to their requirement of fixed sized cells, these algorithms are well suited for the high-throughput, multiple cell configuration.

Since each cell operates independently and can handle multiple data blocks in parallel, TEA and XTEA achieve better throughput on SOSA than on the I-SS or P4. A single cell can perform 175,000 TEA encryptions per second and 170,000 XTEA encryptions per second. Table 12 compares the performance of TEA on different architectures. The table shows that SOSA can achieve 79% of the throughput of the ideal 16-CMP, while using about the same area as a single core with devices switching at a tenth of the speed (1ns vs. 0.1ns). The comparison with I-SOSA highlights the overheads due to simple nodes and limited bandwidth in SOSA.

Table 12. TEA throughput for various architectures

Architecture	Encryptions/sec
P4 @ 3 GHz (100mm^2)	3.9 M/sec
I-SS	73.62 M/sec
16-CMP	1180 M/sec
SOSA (1 cell ~ 0.019mm^2)	0.175 M/sec
I-SOSA (1 cell)	27.7 M/sec
SOSA (5400 cells, 100 mm^2)	940 M/sec
I-SOSA(5400 cells)	72300 M/sec

We have implemented pipelined versions of searching and bin-packing algorithms in SOSA to maximize throughput. Our implementation of search achieves about 10 billion comparisons per second on SOSA while using the same area as a P4 (the P4, I-SS and 16-CMP achieve about 0.5, 2 and 32 billion comparisons per second respectively). We see qualitatively (not quantitatively) similar results for bin-packing. SOSA's ability to exploit data parallelism in these workloads helps it outperform conventional architectures.

Defect Tolerance— The ability to tolerate defects is one of the primary features of SOSA. To test the defect tolerance and to measure the effect of defects on performance, we run a number of experiments varying the node defect rate. Our generated topologies include link defects but these only have an indirect (and minor) effect on performance. Performance is affected if the average number of links per node is less than 2. We find that nodes have 3.2 active links on average. First, we examine the effect of defects on the throughput of a system configured into multiple cells. If we keep the total system area constant (100mm2), as node defect rates increase we are able to configure fewer cells, resulting in reduced throughput. Figure 87 plots the throughput for TEA and XTEA, as node defect rates increase from 0% to 30% revealing a graceful degradation in performance. The connectivity of the random network of nodes is severely affected by node defect rates greater than 30%. This results in network partitions with insufficient functioning nodes in each partition to configure a 64 PE cell.

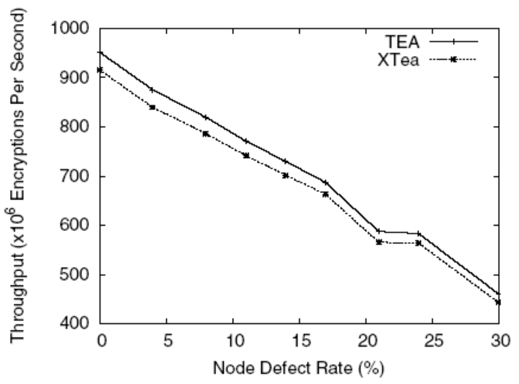


Figure 87: TEA/XTea: Graceful degradation of throughput with increasing node defect rate.

For single cell applications, the entire system must be over-provisioned to ensure that a sufficient number of PEs can be configured. Thus defects indirectly impact performance by reducing network connectivity and bandwidth. In all experiments, SOSA has 30% more nodes (24,000 total nodes) than the minimum needed for a 32x32 matrix multiply. Figure 88 shows the running time for 32x32 matrix multiplication as we increase the number of defective nodes from 0% to 20%. We see that the running time increases by about 8% (compared to a case with no defects), primarily because the average length of PEs increases. We do not present results for the other workloads since they are qualitatively similar. If the system cannot configure sufficient PEs, the problem could potentially be divided into parts that can be solved with the available PEs. Such partitioning, if possible, is beyond the scope of this work. Though the defect tolerance capabilities of the RPF algorithm have been demonstrated before, our experiments show that the ability to tolerate high defect rates incurs only a small performance penalty (~8% for N=32, 32-bit PEs), a characteristic of increasing importance for future systems.

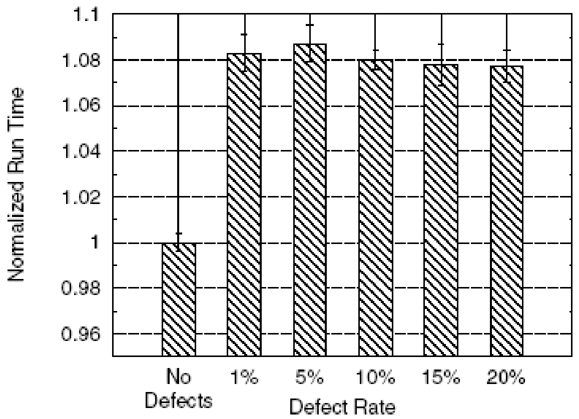


Figure 88: Matrix multiply: The effect of defects on execution time

3.2.6.3 Result Summary

The results in this section show that a system built using a random network of simple nodes can outperform a Pentium 4 (P4) and an ideal superscalar processor (I-SS), despite being severely bandwidth limited and operating devices at a lower switching speed. A scaled up version of the system can outperform an ideal 16-way CMP. The results also highlight SOSA's flexibility in configuring independent cells to improve system utilization and throughput. SOSA provides higher throughput than the P4 and I-SS while using the same area. Coupled with the ability to tolerate a significant defect rate, SOSA shows potential in harnessing the higher device densities that emerging technologies promise to deliver.

3.2.7 Conclusions

With the expected rise in defect rates as device sizes shrink, defect tolerance will be a critical requirement for future system architectures. These increasing defect rates will contribute directly to the exponentially increasing cost of top-down manufacturing. The use of bottom-up techniques like self-assembly will help lower costs but may also result in higher defect rates and a loss of precise control over the manufacturing process. This makes it imperative for architects to develop defect tolerant architectures to exploit the full potential of future nanoscale devices. This report presents SOSA, a self-organizing SIMD architecture built from a random network of simple computational nodes. Despite high defect rates, low bandwidth and lack of underlying physical structure we show that, for data parallel workloads, SOSA is able to perform better than conventional superscalar processors, while operating at a lower speed and consuming much less power. A scaled version of SOSA can perform better than an ideal 16-way CMP. As the underlying technology matures, SOSA's performance can be further improved as fabrication limitations are removed. While SOSA does not solve all problems encountered with self-assembled architectures, it is a step towards realizing defect tolerant computing systems built using emerging technologies that may provide inexpensive terascale integration.

3.2.8 References

[1] H. Abelson et al. Amorphous Computing. Communications of the ACM, 43(5):74–82, 2000.

[2] T. Austin et al. SimpleScalar: An Infrastructure for Computer System Modeling. IEEE Computer, 35(2):59–67, Feb. 2002.

[3] A. Bachtold et al. Logic Circuits with Carbon Nanotube Transistors. Science, 294:1317–1320, Nov. 2001.

[4] P. J. Burke. Carbon Nanotube Devices for GHz to THz Applications. Proc. of SPIE, 5593:52–61, 2004.

[5] S. Ciricescu et al. The Reconfigurable Streaming Vector Processor (RSVP). In Proc. of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, Dec. 2003.

[6] W. B. Culbertson et al. The Teramac Custom Computer: Extending the Limits with Defect Tolerance. In Proc. of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, Nov. 1996.

[7] Y. K. Dalal and R. M. Metcalfe. Reverse Path Forwarding of Broadcast Packets. Communications of the ACM, 21(12):1040–1048, 1978.

[8] A. DeHon. Array-Based Architecture for Molecular Electronics. In Proc. of the First Workshop on Non-Silicon Computation (NSC-1), Feb. 2002.

[9] C. Dwyer et al. DNA Functionalized Single-Walled Carbon Nanotubes. Nanotechnology, 13:601–604, 2002.

[10] C. Dwyer. Self-Assembled Computer Architecture: Design and Fabrication Theory. PhD thesis, University of North Carolina, May 2003.

[11] C.et al. Semi-empirical SPICE Models for Carbon Nanotube FET Logic. In Proc. of the Fourth IEEE Conference on Nanotechnology, Aug. 2004.

[12] R. Espasa et al. Tarantula: A Vector Extension to the Alpha Architecture. In Proc. of the 29th Annual International Symposium on Computer Architecture, May 2002.

[13] S. C. Goldstein and M. Budiu. NanoFabrics: Spatial Computing Using Molecular Electronics. In Proc. of the 28th Annual International Symposium on Computer Architecture, July 2001.

[14] H. Hofstee. Power Efficient Processor Architecture and the Cell Processor. In Proc. of the 11th International Symposium on High-Performance Computer Architecture, pages 258–262, Feb. 2005.

[15] Y. Huang et al. Logic Gates and Computation from Assembled Nanowire Building Blocks. Science, 294:1313–1317, Nov. 2001.

[16] C. Intanagonwiwat et al. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In Mobile Computing and Networking, pages 56–67, 2000.

[17] International Technology Roadmap for Semiconductors, 2005.

[18] U. Kapasi et al. The Imagine Stream Processor. In Proc. 2002 IEEE International Conference on Computer Design, pages 282–288, Sept. 2002.

[19] D. E. Knuth. The Art of Computer Programming. Addison-Wesley, 1973.

[20] C.Leiserson et al. The Network Architecture of the Connection Machine CM-5. In Proc. of the 4th ACM Symposium on Parallel Algorithms and Architectures, pages 272–285, June 1992.

[21] A. Lines. Asynchronous interconnect for synchronous SoC design. IEEE Micro, 24:32–41, Jan/Feb 2004.

[22] R. Lyons and W. Vanderkulk. The Use of Triple-Modular Redundancy to Improve Computer Reliability. IBM Journal, pages 200–209, 1962.

[23] K. Mai et al. Smart Memories: A Modular Reconfigurable Architecture. In Proc. of the 27th Annual International Symposium on Computer Architecture, June 2000.

[24] R. Needham and D. Wheeler. Tea Extensions. Technical report, Computer Laboratory, University of Cambridge, Oct. 1997.

[25] S. H. Park et al. Finite-size, Fully-Addressable DNA Tile Lattices Formed by Hierarchical Assembly Procedures. Angewandte Chemie, 45:735–739, Jan. 2006.

[26] J. P. Patwardhan et al. Circuit and System Architecture for DNA-Guided Self-Assembly of Nanoelectronics. In Foundations of Nanoscience: Self-Assembled Architectures and Devices, pages 344–358, Apr. 2004.

[27] J.Patwardhan et al. Evaluating the Connectivity of Self-Assembled Networks of Nano-scale Processing Elements. In IEEE International Workshop on Design and Test of Defect-Tolerant Nanoscale Architectures (NANOARCH ’05), pages 2.1–2.8, May 2005.

[28] J.Patwardhan et al. Design and Evaluation of Fail-Stop Self-Assembled Nanoscale Processing Elements. In IEEE International Workshop on Design and Test of Defect-Tolerant Nanoscale Architectures (NANOARCH ’06), June 2006.

[29] J.Patwardhan et al. NANA: A Nano-scale Active Network Architecture. ACM Journal on Emerging Technologies in Computing Systems, 2(1):1–30, 2006.

[30] J.Patwardhan et al. Self-Assembled Networks: Control vs. Complexity. In 1st International Conference on Nano-Networks, Sept. 2006.

[31] Performance Database Server. http://www.netlib.org/performance/html/PDStop.html.

[32] B.Robinson and N.Seeman. The design of a biochop: a self-assembling molecular-scale memory device. Protein Engineering, 1:295–300, Aug. 1987.

[33] S. Rosenblatt et al. Mixing at 50GHz using a Single-Walled Carbon Nanotube Transistor. Applied Physics Letters, 87:153111, Oct. 2005.

[34] M. D. Schroeder et al. Autonet: A High-speed, Self-Configuring Local Area Network Using Point to Point Links. IEEE Journal on Selected Areas in Communications, 9(8), Oct. 1991.

[35] K. Skinner et al. Nanowire Transistors, Gate Electrodes, and Their Directed Self-Assembly. In The 72nd Southeastern Section of the American Physical Society (SESAPS), Nov. 2005.

[36] J. von Neumann. Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components. In C.Shannon and J. McCarthy, editors, Automata Studies, pages 43–98. Princeton University Press, Princeton, NJ, 1956.

[37] D. Wheeler and R. Needham. TEA: A Tiny Encryption Algorithm. In Fast Software Encryption: Second International Workshop, Dec. 1994.

[38] E. Winfree et al. Design and Self-Assembly of Two-Dimensional DNA Crystals. Nature, 394:539, 1998.

[39] H. Yan et al. DNA Templated Self-Assembly of Protein Arrays and Highly Conductive Nanowires. Science, 301(5641):1882–1884, Sept. 2003.

4. Final Conclusions

The following sub-sections describe the conclusion and summary-of-results for each component of the project. Each sub-section is a digest of the important findings that resulted from the support of this project.

4.1 Substrate Stabilization and Analysis

DNA nanostructures are the key to building high performance systems beyond photolithography. However, DNA is more reactive than most of the materials used in conventional silicon processing. To achieve the ultimate goal of contextual knowledge mining, DNA nanostructures must be hardened for downstream processing, e.g., functionalization with nanoelectronic or photonic devices.

Psoralen is a molecule that is capable of intercalating into the DNA double helix structure and subsequently cross linking to two bases upon exposure to 365 nm ultraviolet light[49]. Exposure of DNA at lower wavelengths such as 254 nm is known to directly damage the typical helical structure. Studies have also indicated that preferential cross linking occurs more with alternating and adjacent A-T pairs[50]. Although mainly used in the medical field for treatments of skin diseases such as psoriasis and vitiligo, it is also extensively used in biological research concerning the physical properties of DNA helices. Psoralen is a means to improve the integrity of the DNA structures in different environments including a range of temperatures, pH ranges, and chemical environments.

Trioxsalen, or TMP, is a psoralen derivative that is most commonly in medical treatments. As a result, it is one of the more well-studied compounds in its interaction with DNA. It is known that trioxsalen has generally low solubility but is capable of both absorbing wavelengths of light ranging from 250 nm to 350 nm consistently[51]. It is further known that free trioxsalen is capable of fluorescing at 427 nm. Using this information, a range of studies concerning its interaction with addressable DNA grids is being conducted. As this report is only on recent progress, only procedures and results will be presented here.

The solubility of trioxsalen in a set of solvents has been investigated. Literature indicates that standard media for the psoralen derivative include water, buffer, and alcohols. However, previous studies have been demonstrated both in literature and by our group that prolonged exposure to alcohol such as ethanol or isopropanol can destabilize the DNA grid structures[52]. Such data has been quantitatively verified through an experiment in which DNA lattice was exposed to increasing volumes of ethanol ranging from 10% additional volume to 100% additional volume. AFM images were taken and analyzed using image analysis macros developed using ImageJ software. The macros compare the raw image to an ideal DNA lattice image of the same dimensions as well as calculate the percent coverage of DNA material over the image. The two values are used as an index to indicate the extent to which the DNA present is part of the expected structure or how "well-formed" the DNA structures are in the image. Figure 89 presents the results of this experiment.

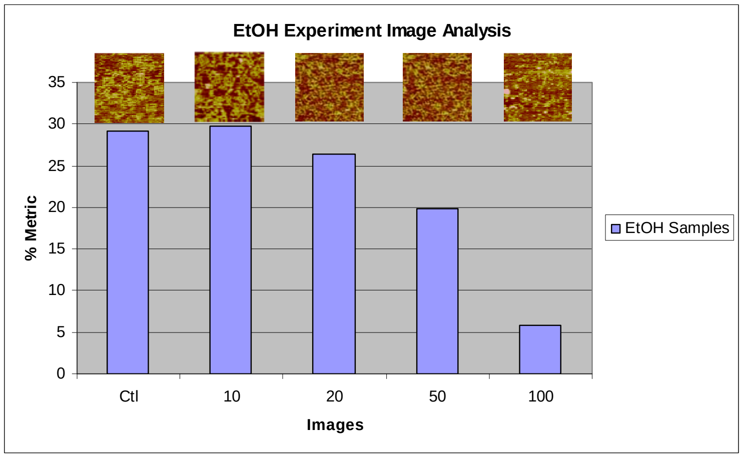


Figure 89: Results from denaturation experiment. The insets above each bar are a section of the AFM image from each sample.

4.2 Scientific Computing on Self-Organized Substrates

The current support for floating point operations on self-organized substrates (SOSA, NANA, etc.) is restricted to software emulation due to physical resource constraints on each node. However, there are a variety of methods to use when implementing a software floating point library and we have begun to investigate these various tradeoffs for SOSA. Our current work has identified floating point renormalization as a severe bottleneck in system performance. For example, SOSA integer-only execution time of a mixture of 50 multiplications and 50 additions approaches 360μs while an identical set of floating point operations takes nearly 13ms, or almost 40-times longer for the floating point operations.

Clearly, for SOSA to find practical application to any scientific domain either (i) fixed-point (integer-only) operations must suffice, or (ii) floating point operation performance must be improved.

The identification of the renormalization step in the software floating point operations focused our efforts on reducing this cost. In fact, renormalization has a long history of optimizations and prior to the adoption of the IEEE floating point standard was largely customizable per application. The approach we adopt here is to use a high-radix exponent to help reduce the time spent renormalizing operands. Since a floating point number can be represented as N = Mantissa * (Base ^ Exponent) we are free to choose a value for the Base. The IEEE standard defines the base as 2 which requires O(n) shifts to normalize a number where 'n' is the number of bits in the mantissa. For our SOSA implementation we have evaluated several bases, as shown in Table 13, to find an optimal design point at base 16.

Table 13. High-radix floating point operation execution time

Base	Exe. time (μs)
2	13027.850
4	2864.058
8	1784.806
16	1459.502
32	2797.878
Integer-only	352.817

The improvement of base-16 exponents over base-2 is primarily due to the reduction in time for renormalization on each addition. This yields a 9X improvement for base-16 over base-2 which brings the SOSA floating point performance within a factor of 4X of integer-only performance (vs. ~50X previously.) We have also begun to study the impact of the size of processing elements and the number of nodes per element to understand how this might be used to further improve floating point performance. We are also investigating two new architectural innovations focused on reducing shift-by-operand latency and improvements in the methods used to self-organize the interconnect.

The SOSA system is built using a random network of simple nodes can outperform a Pentium 4 (P4) and an ideal superscalar processor (I-SS), despite being severely bandwidth limited and operating devices at a lower switching speed. We have shown that floating point performance can be improved by 9X over the IEEE standard implementation and comes within 4X of integer-only performance. With the expected rise in defect rates as device sizes shrink, defect tolerance will be a critical requirement for future system architectures. These increasing defect rates will contribute directly to the exponentially increasing cost of top-down manufacturing. The use of bottom-up techniques like self-assembly will help lower costs but may also result in higher defect rates and a loss of precise control over the manufacturing process. This makes it imperative for architects to develop defect tolerant architectures to exploit the full potential of future nanoscale devices.

4.3 Defect Tolerant Computer Architecture Results

The continual decrease in transistor size (through either scaled CMOS or emerging nano-technologies) promises to usher in an era of tera to peta-scale integration. However, this decrease in size is also likely to increase defect densities, contributing to the exponentially increasing cost of top-down lithography. Bottom-up manufacturing techniques, like self-assembly, may provide a viable lower-cost alternative to top-down lithography, but may also be prone to higher defects. Therefore, regardless of fabrication methodology, defect tolerant architectures are necessary to exploit the full potential of future increased device densities.

This report explores a defect tolerant SIMD architecture. A key feature of our design is the ability of a large number of limited capability nodes with high defect rates (up to 30%) to self-organize into a set of SIMD processing elements. Despite node simplicity and high defect rates, we show that by supporting the familiar data parallel programming model the architecture can execute a variety of programs. The architecture efficiently exploits a large number of nodes and higher device densities to keep device switching speeds and power density low. On a medium sized system (~1cm^2 area), the performance of the proposed architecture on our data parallel programs matches or exceeds the performance of an aggressively scaled out-of-order processor (128-wide, 8k reorder buffer, perfect memory system). For larger systems (>1cm^2), the proposed architecture can match the performance of a chip multiprocessor with 16 aggressively scaled out-of-order cores.

Manufacturing defects, power density, process variability, transient faults, bulk silicon limits, rising test costs and multibillion dollar fabrication facilities are some of the challenges facing the continued scaling of CMOS. While architectural modifications (e.g., multicore) can provide some short-term relief, the semiconductor industry recognizes the importance of these issues and the need to explore long term alternatives to CMOS devices and fabrication techniques [17].

One promising alternative is DNA-based self-assembly of nanoscale components using inexpensive laboratory equipment to achieve tera to peta-scale integration. Although much of this technology is in its infancy (i.e., demonstrated in research lab experiments), by studying its potential uses for building computing systems, architects can gain a deeper understanding of its limitations and opportunities while providing important feedback to the scientists developing the new technologies.

DNA-based fabrication produces precise control within a small area (e.g., 9 μm^2) enabling the construction of a large number (~10^9-10^{12}) of small nodes (computational circuits with ~10^4 transistors) that can be linked together using self-assembly. This produces a random network of nodes, due to the lack of control over placement and orientation of nodes, which contain defective nodes and links. While our work is motivated by DNA-based self-assembly, it is applicable to any technology with similar characteristics (e.g., scaled CMOS with high process variability, high defect rates and point-to-point links between relatively small compute nodes). The challenge for computer architects is to efficiently exploit the computational power of the large number of nodes while overcoming two primary challenges: 1) loss of precise control over the entire fabrication process, and 2) high defect rates.

This report presents a SIMD architecture designed to address these challenges. The fundamental building block in our architecture is a relatively small node (e.g., 1-bit ALU with 32 bits of storage and communication support for four neighbors) that operates asynchronously. A configuration phase at startup isolates defective nodes and allows groups of nodes to self-organize into SIMD processing elements (PEs) which are connected in a logical ring, thus simplifying the programmer's view of the system.

Simulations using conservative estimates for node size and device speed show that the proposed design can match the performance of aggressively scaled architectures for 8 out of 9 benchmarks tested. Furthermore, this performance is achieved with a very low power density of 6.5 W/cm^2 (vs. >75 W/cm^2 for modern cores) while conservatively assuming that about 90% of the devices in the system switch every nanosecond. Finally, we show that our system can tolerate up to 30% defective nodes. Our results demonstrate the potential of this technology for building high performance architectures despite high defect rates and loss of precise control during fabrication. Further improvements are possible as the technology scales to allow more complex nodes, better inter-node connectivity, and faster devices. Our main contributions are:

- adapting self-organization methods to computer architectures,
- designing a node that balances fabrication constraints with functionality needed to communicate, compute, and self-organize, and,

- demonstrating the above capabilities by composing a high-performance, defect tolerant SIMD architecture from a random network of nodes.

4.4 Advances in DNA Nanogrid Synthesis

The hierarchical construction of DNA nanostructures has enabled larger and more complex designs than previous demonstrations. Coupled with good thermodynamic design and thermal ordering, hierarchies of interactions can be used to assemble complex chemical patterns. We have recently demonstrated that our original hierarchy can be extended to build 8x8 (and 8x16) grids of DNA as shown in Figure 90.

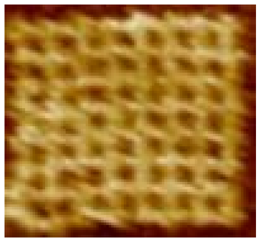
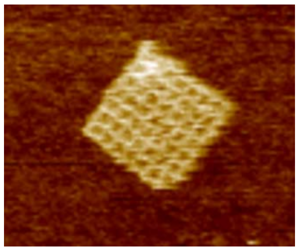
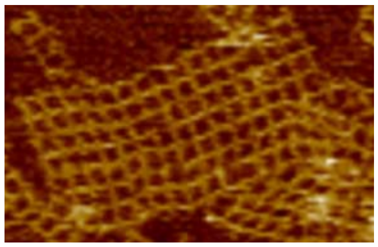
(a)  (b)  (c) 

Figure 90: (a), (b) Two individual 8x8 DNA grids. Each grid is 140nm x 140nm. (c) An un-optimized 8x16 DNA grid built using two 8x8 grids.

These methods have enabled large structures with aperiodic chemical patterns. Currently, we have biotin-streptavidin patterns (i.e., protein patterns) and preliminary demonstrations of Ag nanoparticle patterns. Our strategy to nucleate Ag nanoparticles has been to use a templating protein (e.g., streptavidin) that has been chemically pre-charged with precursor reagents. After lengthy optimization, we have found an operating point where the protein retains a strong binding affinity to the biotin-patterned DNA grids, yet should nucleate Ag nanoparticles. The benefit of this approach is that the surfaces of the patterned metallic particles are unfettered by solubilization or stabilization groups (e.g., citrate). That is, protein nucleated Ag nanoparticles may yield "cleaner" particle structures than otherwise possible.

Toward this end we have synthesized several wire-like structures from protein-biotin-grid interactions that appear in typical logic and memory structures. Figure 91 shows fours AFM images of the wire structures and this demonstrates that it is possible to synthesize, in quantities approaching 10^{12}, particle patterns with useful aperiodicity.

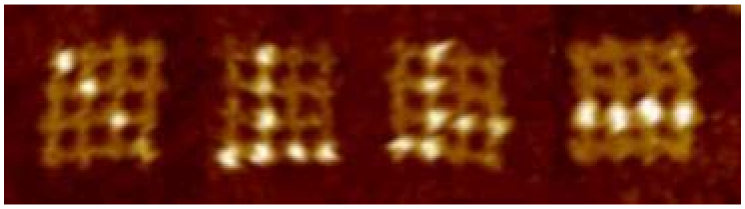


Figure 91: Four patterned DNA grids. Each image represents a wire-like motif that can be found in typical logic circuitry.

This work is on-going and will further study the nucleation of Ag (and semiconductor) nanoparticles on the grids as well as the controlled fusion of core-shell particles on the grids to form nanoelectronic junctions by shell over-growth. We will also continue to investigate the opportunity of optical resonance energy transfer mediated by DNA grid nanostructures. We have preliminary data that suggests it is possible to form near-field conduits for optical excitons.

4.5 DNA Self-assembly CAD Results

The continued scaling of CMOS and the emergence of alternative nano-electronic devices promises to deliver increased device densities and higher operating frequencies. However, small devices are more vulnerable to defects, contributing directly to the exponentially increasing costs of top-down lithography. DNA-based self-assembly of nano-electronic devices is one approach that has potential as a lower-cost long-term replacement of conventional top-down fabrication techniques. Although much of this technology is in its infancy (i.e., demonstrated in research lab experiments), by studying its potential uses for building computing systems architects can gain a deeper understanding of its limitations and opportunities while providing important feedback to the scientists developing the new technologies. The precise binding rules of DNA enable creation of nanostructures with minimum pitch on the order of a few nanometers. These nanostructures can be used to place and interconnect nanoscale components (e.g., crossed carbon nanotube FETs, ring-gated FETs, nanowires).

The challenge in creating DNA nanostructures is to specify the appropriate DNA sequences such that the desired structure (geometry) forms and is thermodynamically stable. To meet this challenge, DNA self-assembly can exploit the common technique of composing a small set of relatively simple motifs to create more sophisticated structures. Many parts of this design process can benefit from design automation. However, in this work we focus on the key aspect of designing the DNA sequences that control how motifs can bind with each other. Specifically, we seek to find DNA sequences that minimize the strength of unintentional interactions with the other motifs in the set while maximizing the strength of intentional interactions.

Our approach is to evaluate the sequence design space to create a fixed size 60nm X 60nm grid with 20nm pitch. This structure is composed through a hierarchical assembly of motifs. We focus on the design of the final assembly step that combines 16 cruciform motifs (arranged 4x4) to form the final grid structure. For this structure, we must determine the best 96 sequences that satisfy the structural and stability metrics. To accomplish this we implemented an optimization algorithm that is aware of both intentional and unintentional interactions and exploits parallelism to rapidly evaluate the large sequence design space.

We have experimentally verified our method by designing, synthesizing, and assembling the target nanostructure and characterizing it with atomic force microscopy (AFM). We also show that our optimization algorithm produces superior sequences for a 2x2 grid than sequences produced using conventional text-based sequence comparison or random sequence selection.

4.6 Computer Architecture Survey Results

The continued scaling of CMOS and the emergence of alternative nano-electronic devices promises to deliver increased device densities and higher operating frequencies. However, small devices are more vulnerable to defects, contributing directly to the exponentially increasing costs of top-down lithography. DNA-based self-assembly of nano-electronic devices is one approach that has potential as a lower-cost long-term replacement of conventional top-down fabrication techniques. Although much of this technology is in its infancy (i.e., demonstrated in research lab experiments), by studying its potential uses for building computing systems architects can gain a deeper understanding of its limitations and opportunities while providing important feedback to the scientists developing the new technologies. This fabrication approach produces precise control within a small area (e.g., 9 μm^2) enabling the construction of a large number of (~10^9-10^{12}) small computational circuits (nodes) that can be linked to each other using self-assembly. This results in a random network of nodes that contains defective nodes and links, due to the lack of control over placement and orientation of the nodes. While our work is motivated by DNA-based self-assembly, it is applicable to any technology with similar characteristics (e.g., scaled CMOS with high process variability, high defect rates and point-to-point links between relatively small compute nodes). The challenge for computer architects is to efficiently exploit the computational power of this random network while overcoming two primary challenges: 1) loss of precise control over the entire fabrication process, and 2) high defect rates.

Self-assembly of nano-electronic devices has the potential to emerge as a lower cost alternative to top-down manufacturing. DNA-based self-assembly, is a bottom-up manufacturing process that uses the precise binding rules of DNA with nanoscale devices to build computing systems. We assume a proposed assembly process to place electronic circuits on a DNA lattice. A key requirement of this process is the ability to control the placement of electronic devices (e.g., carbon nanotubes or silicon nanowires) at specific points on the 60nm X 20nm circuit. Recently, we have taken two critical steps towards this goal by demonstrating the placement of aperiodic patterns, with a 20nm pitch, on a DNA lattice and the DNA-guided self-assembly of nanowire transistors.

Current self-assembly processes produce limited size DNA lattices thus limiting circuit size. However, the parallel nature of self-assembly enables the construction of a large number (~10^9-10^{12}) of nodes that may be linked together by self-assembled conducting nanowires. Self-assembly does not control the placement and orientation of nodes as they are interconnected resulting in the formation of a random network of nodes that contains defective nodes and links. Communication with external CMOS circuitry occurs through a metal junction (“via”) that overlaps several nodes but interfaces with the network of nodes through a single “anchor node”. There may be several via/anchor node pairs in large networks.

A computing system built from a random network must: a) tolerate node and interconnect defects, b) not rely on underlying network structure, c) compose more powerful computational blocks from simple nodes, d) minimize communication overheads, and e) achieve performance that is at least comparable to future CMOS based systems. Several research projects examined building computing systems with a subset of these goals, including self configuration, routing and resiliency in the face of defects and the ability to compose complex computational units from simpler blocks, but we face added challenges because of the extremely limited computational capabilities available in nodes.

The results from this work are a defect tolerant computing system using a random network of nodes that achieves performance comparable to future CMOS based systems. To efficiently utilize large numbers ($>10^9$-10^{12}) of nodes we implement a SIMD architecture and focus on data parallel workloads. Our proposed system - called the “Self-assembled SIMD Architecture” (SSA) - supports a three operand register-based ISA with predicated execution and explicit PE-Shift instructions to move data between PEs and communicate with an external controller. We assume that the external controller has access to a conventional memory system. Each self-assembled node is a fully asynchronous circuit and there is no global clock to synchronize data transfers between or within nodes. Each node has a 1-bit ALU with a small register file, and nodes are connected to each other by single wire links. Each link supports very low bandwidth asynchronous communication that transfers 1 bit of data per handshake. To support deadlock-free routing, we add support for three virtual channels (1 bit each). The random network of nodes is organized at two levels during a configuration phase. First, since a node is too small to hold a PE, we group sets of nodes to form a PE. Second, PEs are linked in a logical ring providing programmers a simplified system view to reason about inter-PE communication. Configuration, initiated from an anchor, maps out defective nodes and connects functional nodes in a broadcast tree. The system can be configured in two ways: a) as a monolithic system, all nodes on one logical ring (one “cell”), or b) as multiple, independent logical rings (multiple “cells”). For a monolithic system, anchors can be used to speed up PE configuration and data input/output by serving as “taps” into the logical ring. The only constraint enforced during configuration is that an anchor cannot partition a PE. In case (b), we achieve space partitioning by running the configuration algorithm from multiple anchors to create independent cells. Space partitioning is a common technique used in highly parallel systems to increase resource utilization by enabling the execution of multiple instances of one workload, or running multiple workloads.

4.7 Data Mining Results

Data mining is the process of extracting or mining interesting knowledge from large amounts of data stored in databases or other information stores. A fundamental and essential problem in data mining is to discover frequent patterns or itemsets: given a data base of item transactions, find all itemsets that occur in at least a user-specified percentage of the total transactions in the database (i.e., has a specified support level). Frequent itemset mining is useful for many data mining capabilities such as discovery of association rules [1], strong rules, correlations, sequential rules [2], episodes [12], multi-dimensional patterns, partial periodicity [9], and many other important discovery tasks. These data mining capabilities are useful for practical applications such as market basket analysis, inferring patterns from web access logs, and network intrusion detection among others.

Frequent itemset mining generates a very large number of patterns which may reduce not only the efficiency but also effectiveness of mining, since it generates numerous redundant patterns. Furthermore, users must sift through a large number of mined patterns to find useful ones. Closed frequent itemset (CFI) mining [14], mines only those frequent itemsets having no proper superset with the same support, which can lead to orders of magnitude smaller result set than mining frequent itemsets [15]. CFI mining retains all the information of frequent itemset mining, as it is straightforward to generate all the frequent itemsets from the closed frequent itemsets.

We have explored the memory hierarchy performance of a state-of-the-art CFI algorithm—called FPclose[1] [8]—using real world datasets. We used a combination of tools (OProfile [11], VTUNE [16], Simplescalar and iostat) to gain insight into FPclose’s memory hierarchy performance for both sparse and dense real world datasets on both Pentium III and Pentium 4 systems. The characterization results indicate that cache behavior becomes a critical performance bottleneck as the support level decreases. Simulation experiments show that we achieve gains in the IPC (Instructions per cycle) for a system with perfect memory hierarchy, thus the algorithm is memory bound. However, an increase in the width of the instruction window has almost no effect on the IPC.

One specific function CFI_tree::insert(bool*,…) accounts for up to 69% of the L1 data cache misses, 86% of the L2 data cache misses, and 69% of the execution time for moderate to low support levels. This function is unique to the closed frequent itemset problem. Symbolic profiling and source code inspection reveal that CFI_tree::insert(bool*,…) function incurs most of its cache misses while traversing the nodes of a prefix tree (or closed frequent pattern tree) at a specific level (right sibling pointer dereferences). This motivates the application of well-known data structure modifications to improve cache performance: 1) padding & aligning, and 2) node clustering [4]. Our characterization results also reveal that the average number of nodes accessed during the sibling traversal is data dependent. This motivates a final optimization that dynamically increases node clustering as more nodes are allocated in a specific level of the prefix tree.

We evaluated the data structure modifications by measuring execution cycles on Pentium III and Pentium 4 systems and compared with execution cycles of our base case. Our results show that padding and aligning have no significant impact on performance. Node clustering achieves performance gains of up to 79% on Pentium III systems and 65% on Pentium 4. Finally, we show that the dynamic allocation technique provides speedups up to 81% on Pentium III system and 67% on Pentium 4. The maximum difference in speedup of the best static sized node clustering scheme at each data point and the dynamic scheme was 6% on Pentium III and 8% on Pentium 4. This difference was in dense datasets, for sparse datasets dynamic scheme performed as well as or better than the best static sized node clustering scheme at each data point in many cases for both systems. For dense datasets node clustering produces the greatest improvements in performance on both Pentium systems.

4.8 References

[1] A. Chworos, I. Severcan, A. Y. Koyfman, P. Weinkam, E. Oroudjev, H. G. Hansma, and L. Jaeger, "Building programmable jigsaw puzzles with RNA," *Science*, vol. 306 (5704), 2004.

[2] A. Y. Koyfman, G. Braun, S. Magonov, A. Chworos, N. O. Reich, and L. Jaeger, "Controlled Spacing of Cationic Gold Nanoparticles by Nanocrown RNA," *Journal of the American Chemical Society*, vol. 127 (34), pp. 11886-11887, 2005.

[3] C. Dwyer, S. H. Park, T. H. LaBean, and A. R. Lebeck, "The Design and Fabrication of a Fully Addressable 8-tile DNA Lattice," Proceedings of the Foundations of Nanoscience: Self-Assembled Architectures and Devices, 2005.

[4] S. H. Park, C. Pistol, S. J. Ahn, J. H. Reif, A. R. Lebeck, C. Dwyer, and T. H. LaBean, "Finite-size, Fully-Addressable DNA Tile Lattices Formed by Hierarchical Assembly Procedures," *Angewandte Chemie*, vol. 45, pp. 735-739, 2006.

[5] C. Pistol, A. R. Lebeck, and C. Dwyer, "Design Automation for DNA Self-Assembled Nanostructures," Proceedings of the 43rd Design Automation Conference (DAC), 2006.

[6] P. W. K. Rothemund, "Folding DNA to create nanoscale shapes and patterns," *Nature*, vol. 440 (7082), pp. 297-302, 2006.

[7] N. C. Seeman, "Nucleic Acid Junctions and Lattices," *Journal of Theoretical Biology*, vol. 99, pp. 237-247, 1982.

[8] N. C. Seeman, "DNA Engineering and its Application to Nanotechnology," *Trends in Biotechnology*, vol. 17, pp. 437-443, 1999.

[9] R. P. Goodman, I. A. T. Schaap, C. F. Tardin, C. M. Erben, R. M. Berry, C. F. Schmidt, and A. J. Turberfield, "Rapid Chiral Assembly of Rigid DNA Building Blocks for Molecular anofabrication," *Science*, vol. 310 (5754), pp. 1661-1665, 2005.

[10] N. C. Seeman, H. Wang, B. Liu, J. Qi, X. Li, X. Yang, F. Liu, W. Sun, Z. Shen, R. Sha, C. Mao, Y. Wang, S. Zhang, T. J. Fu, S. Du, J. E. Mueller, Y. Zhang, and J. Chen, "The Perils of Polynucleotides: The Experimental gap Between the Design and Assembly of Unusual DNA Structures," Proceedings of the Second International Meeting on DNA Based Computers (DNA2), 1996.

[11] C. Mao, W. Sun, and N. C. Seeman, "Designed Two-Dimensional DNA Holliday Junction Arrays Visualized by Atomic Force Microscopy," *Journal of the American Chemical Society*, vol. 121, pp. 5437-5443, 1999.

[12] J. Sharma, R. Chhabra, Y. Liu, Y. G. Ke, and H. Yan, "DNA-templated Self-assembly of Two-dimensional and Periodical Gold," *Angewandte Chemie-international Edition*, vol. 45 (5), pp. 730-735, 2006.

[13] D. Reishus, B. Shaw, Y. Brun, N. Chelyapov, and L. Adleman, "Self-assembly of DNA Double-double Crossover Complexes into Hiigh-density, Doubly Connected, Planar Structures," *Journal of the American Chemical Society*, vol. 127 (50), pp. 17590-17591, 2005.

[14] H. Yan, S. H. Park, G. Finkelstein, J. H. Reif, and T. H. LaBean, "DNA Templated Self-Assembly of Protein Arrays and Highly Conductive Nanowires," *Science*, vol. 301, pp. 1882-1884, 2003.

[15] Y. He, Y. Chen, H. P. Liu, A. E. Ribbe, and C. D. Mao, "Self-assembly of Hexagonal DNA Two-dimensional (2D) Arrays," *Journal of the American Chemical Society*, vol. 127 (35), pp. 12202-12203, 2005.

[16] J. W. Zheng, P. E. Constantinou, C. Micheel, A. P. Alivisatos, R. A. Kiehl, and N. C. Seeman, "Two-dimensional Nanoparticle Arrays Show the Organizational Power of Robust DNA Motifs," *Nano Letters*, vol. 6 (7), pp. 1502-1504, 2006.

[17] E. Winfree, F. Liu, L. A. Wenzler, and N. C. Seeman, "Design and Self-Assembly of Two-Dimensional DNA Crystals," *Nature*, vol. 394, pp. 539-544, 1998.

[18] R. D. Barish, P. W. K. Rothemund, and E. Winfree, "Two computational primitives for algorithmic self-assembly: Copying and counting," *Nano Letters*, vol. 5 (12), pp. 2586-2592, 2005.

[19] C. Pistol and C. Dwyer, "Scalable, low-cost, hierarchical assembly of programmable DNA nanostructures," *Nanotechnology*, vol. 18, pp. 125305-9, 2007.

[20] G. M. Whitesides and B. A. Grzybowski, "Self-Assembly at All Scales," *Science*, vol. 295, pp. 2418-2421, 2002.

[21] B. Valeur, *Molecular Fluorescence: Principles and Applications*. Weinheim: Wiley-VCH, 2002.

[22] Y. Ohya, K. Yabuki, M. Hashimoto, A. Nakajima, and T. Ouchi, "Multistep Fluorescence Resonance Energy Transfer in Sequential Chromophore Array Constructed on Oligo-DNA Assemblies," *Bioconjugate Chemistry*, vol. 14, pp. 1057-1066, 2003.

[23] L. R. Dalton, A. W. Harper, and B. H. Robinson, "The role of London forces in defining noncentrosymmetric order of high dipole moment*high hyperpolarizability chromophores in electrically poled polymeric thin films," *Proceedings of the National Academy of Sciences*, vol. 94, pp. 4842-4847, 1997.

[24] A. W. Harper, S. Sun, L. R. Dalton, S. M. Garner, A. Chen, S. Kalluri, W. H. Steier, and B. H. Robinson, "Translating Microscopic Optical Nonlinearity into Macroscopic Optical Nonlinearity: the Role of Chromophore-chromophore Electrostatic Interactions," *Journal of the Optical Society of America B-optical Physics*, vol. 15 (1), pp. 329-337, 1998.

[25] J. Shi and D. E. Bergstrom, "Assembly of Novel DNA Cycles with Rigid Tetrahedral Linkers," *Angewandte Chemie International Edition*, vol. 36 (1-2), pp. 111-113, 1997.

[26] J. R. Lakowicz, *Principles of Fluorescence Spectroscopy*. New York: Kluwer Academic / Plenum Publishers, 1999.

[27] D. W. Brousmiche, J. M. Serin, J. M. J. Fre©*chet, G. S. He, T.-C. Lin, S. J. Chung, and P. N. Prasad, "Fluorescence Resonance Energy Transfer in a Novel Two-Photon Absorbing System," in *Journal of the American Chemical Society*, vol. 125, 2003, pp. 1448-1449.

[28] J. P. Patwardhan, C. Dwyer, A. R. Lebeck, and D. J. Sorin, "NANA: a Nano-scale Active Network Architecture," *J. Emerg. Technol. Comput. Syst.*, vol. 2 (1), pp. 1-30, 2006.

[29] J. P. Patwardhan, V. Johri, C. Dwyer, and A. R. Lebeck, "A Defect Tolerant Self-organizing Nanoscale SIMD Architecture," Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, 2006.

[30] Stmicroelectronics, "ST6200C/ST6201C/ST6203C Datasheet," 2007.

[31] S. Freescale, "MC9RS08KA2, MC9RS08KA2 Datasheet," 2007.

[32] D. Endy, "Foundations for Engineering Biology," *Nature*, vol. 438 (7067), pp. 449-453, 2005.

[33] A. P. Arkin and D. A. Fletcher, "Fast, Cheap and Somewhat in Control," *Genome Biology*, vol. 7 (8), pp. 114.1-114.6, 2006.

[34] I. Tinoco, K. Sauer, and J. C. Wang, *Physical Chemisty: Principles and Applications in Biological Science*. Upper Saddle River, NJ: Prentice Hall, 762 pgs., 1995.

[35] C. Berney and G. Danuser, "FRET or No FRET: A Quantitative Comparison," *Biophysical Journal*, vol. 84, pp. 3992-4010, 2003.

[36] C. Dwyer, J. Poulton, R. M. Taylor, and L. Vicci, "DNA Self-assembled Parallel Computer Architectures," *Nanotechnology*, vol. 15, pp. 1688-1694, 2004.

[37] M. T. Niemier and P. M. Kogge, "Exploring and Exploiting Wire-Level Pipelining in Emerging Technologies," Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01), 2001.

[38] S. C. Goldstein and M. Budiu, "NanoFabrics: Spatial Computing Using Molecular Electronics," Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA), 2001.

[39] M. G. Ancona, "Systolic Processor Designs Using Single-Electron Digital Circuits," in *Superlattices and Microstructures*, vol. 20, 1996, pp. 461-472.

[40] P. Beckett and A. Jennings, "Toward Nanocomputer Architecture," in *Seventh Asia-Pacific Computer Systems Architecture Conference*, 2002, pp. 141-150.

[41] T. J. Fountain, M. J. B. Duff, D. G. Crawley, C. D. Tomlinson, and C. D. Moffat, "The Use of Nanoelectronic Devices in Highly-Parallel Computing Systems," in *IEEE Transactions on VLSI Systems*, vol. 6, 1998, pp. 31-38.

[42] A. DeHon, "Array-based Architecture for Fet-based, Nanoscale Electronics," *IEEE Transactions on Nanotechnology*, vol. 2 (1), pp. 23 - 32, 2003.

[43] D. Copsey, M. Oskin, F. Impens, T. Metodiev, A. Cross, F. T. Chong, I. L. Chuang, and J. Kubiatowicz, "Toward a Scalable, Silicon-based Quantum Computing Architecture," *IEEE Journal of Selected Topics in Quantum Electronics*, vol. 9 (6), pp. 1552-1569, 2003.

[44] N. Isailovic, Y. Patel, M. Whitney, and J. Kubiatowicz, "Interconnection Networks for Scalable Quantum Computers," Proceedings of the 33rd Annual International Symposium on Computer Architecture, 2006.

[45] M. Oskin, F. T. Chong, I. L. Chuang, and J. Kubiatowicz, "Building Quantum Wires: the Long and the Short of It," Proceedings of the 30th Annual International Symposium on Computer Architecture, 2003.

[46] A. Huang, "Optical Digital Computers," Proceedings of the 1989 Acm/ieee Conference on Supercomputing, 1989.

[47] V. P. Heuring, H. F. Jordan, and P. Pratt, "Bit Serial Optical Computer Design," Proceedings of the Spie, 1988.

[48] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. F. Knight, R. Nagpal, E. Rauch, G. J. Sussman, and R. Weiss, "Amorphous Computing," in *Cacm*, vol. 43, 2000, pp. 74-82.

[49] C. V. Hanson, C.-k. J. Shen, and J. E. Hearst, "Cross-Linking of DNA in situ as a Probe for Chromatin Structure," *Science*, vol. 193 (4247), pp. 62-64, 1976.

[50] M. Ramaswamy and A. T. Yeung, "The Reactivity of 4,5',8-trimethylpsoralen with Oligonucleotides Containing AT Sites," *Biochemistry*, vol. 33 (18), pp. 5411-5413, 1994.

[51] A. Losi, R. Bedotti, and C. Viappiani, "Time-Resolved Photoacoustics Determination of Intersystem Crossing and Singlet Oxygen Photosensitization Quantum Yields for 4,5',8-Trimethylpsoralen," *Journal of Physical Chemistry*, vol. 99 (43), pp. 16162-16167, 1995.

[52] J. Piskur and A. Rupprecht, "Aggregated DNA in Ethanol Solution," *Febs Letters*, vol. 375 (3), pp. 174-178, 1995.