# Dynamic Access Ordering for Symmetric Shared-Memory Multiprocessors

Sally A. McKee

Computer Science Report No. CS-94-14
May, 1994

# Dynamic Access Ordering for Symmetric Shared-Memory Multiprocessors

Sally A. McKee
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
mckee@cs.virginia.edu

## Abstract

Memory bandwidth is rapidly becoming the performance bottleneck in the application of high performance microprocessors to vector-like algorithms, including the "Grand Challenge" scientific problems. Caching is not the sole solution for these applications due to the poor temporal and spatial locality of their data accesses. Moreover, the nature of memories themselves has changed. Achieving greater bandwidth requires exploiting the characteristics of memory components "on the other side of the cache" — they should not be treated as uniform access-time RAM.

This paper describes the use of hardware-assisted *access ordering* in symmetric multiprocessor (SMP) systems. Our technique combines compile-time detection of memory access patterns with a memory subsystem (called a *Stream Memory Controller*, or SMC) that decouples the order of requests generated by the computational elements from that issued to the memory system. This decoupling permits the requests to be issued in an order that optimizes use of the memory system. Our simulations indicate that SMP SMC systems can consistently deliver nearly the full system bandwidth.

# Dynamic Access Ordering for Symmetric Shared-Memory Multiprocessors

## 1. Introduction

Processor speeds are increasing much faster than memory speeds [Kat89, Hen90]. As a result, memory bandwidth is rapidly becoming the limiting performance factor for many applications. A comprehensive, successful solution to the memory bandwidth problem must exploit the richness of the *full* memory hierarchy. This requires not only finding ways to improve cache performance, but providing alternatives for computations for which caching is insufficient.

Most memory-bandwidth studies focus on cache hit rates, but these only address one aspect of the problem. Most memory devices manufactured in the last decade provide special capabilities that make it possible to perform some access sequences faster than others [IEE92, Ram92, Qui91], and exploiting these component characteristics can dramatically improve effective bandwidth. For applications that perform vector-like memory accesses, for instance, bandwidth can be increased by reordering the requests to take advantage of device properties such as fast-page mode.[1] This *access ordering* is straightforward to apply, and requires no heroic compiler technology. Access ordering yields a dramatic performance improvement over traditional caching of vector operands, especially for non-unit stride vector computations.

Access ordering may be performed statically, at compile time, or dynamically, at run time. Moyer derives compile-time access-ordering algorithms relative to a precise analytic model of memory systems [Moy93]. This approach unrolls loops and orders non-caching memory operations to exploit architectural and device features of the target memory

1. These devices behave as if implemented with a single on-chip cache line, or *page*. A memory access falling outside the address range of the current DRAM page forces a new page to be set up. The overhead time required to do this makes performing such an access significantly slower than one that hits the current page.

system. McKee et. al. propose a uniprocessor architecture for performing access ordering at run time [McK94a]. Simulation studies indicate that dynamic access ordering is a valuable technique for improving uniprocessor memory performance for stream computations — the SMC, or *Stream Memory Controller*, consistently delivers almost the entire available bandwidth [McK93a, McK94b, McK93c].

The applicability of dynamic access ordering is not limited to uniprocessor environments. This paper discusses the effectiveness of dynamic access ordering with respect to the memory performance of symmetric multiprocessor (SMP) systems. Our simulation results show that a modest number of computational elements (CEs) sharing an SMC can achieve a similar percentage of peak bandwidth as a uniprocessor SMC system.

## 2. The Stream Memory Controller

Access ordering systems span a spectrum ranging from purely compile-time to purely run-time approaches. A general taxonomy of access ordering systems is presented in [McK93a]. Based on our analysis and simulations, we believe that the best engineering choice is to detect streams at compile time, and to defer access ordering and issue to run time. Here we describe in general terms how such a scheme might be incorporated into an overall system architecture, first for a uniprocessor system and then for a symmetric shared-memory multiprocessor system.

The approach we suggest will be described based on the simplified architectures of Figure 1 and Figure 2. In the uniprocessor system depicted in Figure 1, memory is interfaced to the computational element, or “CE”, through a controller labeled “MSU” for Memory Scheduling Unit. The MSU includes logic to issue memory requests as well as logic to determine the order of requests during streaming computations. For non-stream accesses, the MSU provides the same functionality and performance as a traditional memory

controller. This is crucial — the access-ordering circuitry of the MSU is *not* in the critical path to memory and in no way affects scalar processing.

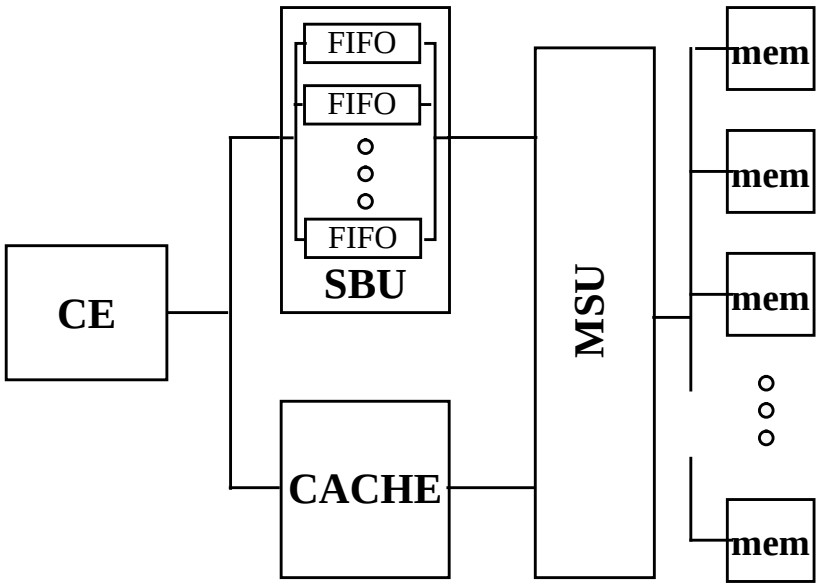



**Figure 1 Uniprocessor SMC Organization**

The MSU has full knowledge of all streams currently needed by the processor: given the base address, vector stride, and vector length, it can generate the addresses of all elements in a stream. The scheduling unit also knows the details of the memory architecture, including interleaving and device characteristics. The access-ordering circuitry uses this information to issue requests for individual stream elements in an order that attempts to optimize memory system performance.

A separate Stream Buffer Unit (SBU) provides memory-mapped control registers that the processor uses to specify stream parameters and high-speed buffers for stream operands. These buffers are implemented as FIFOs that are asynchronously filled from (or drained to) memory by the access/issue logic. Each stream of the computation is assigned to one FIFO, and load instructions from (or store instructions to) a particular stream reference the FIFO head via a memory-mapped register. As with the stream-specific parts of the MSU, the SBU is not on the critical path to memory, and the speed of non-vector accesses is not adversely

affected by its presence. Together, the MSU and SBU comprise a Stream Memory Controller (SMC) system.

When adapting this general framework to an SMP system, a number of options exist regarding placement of SMC components, depending on where we draw chip boundaries. The most efficient organization is one in which the entire SMC system and all computational elements reside on a single chip; this is the organization we consider here. If a single-chip implementation is not feasible, several possibilities remain. Placing a full SMC system on each processor chip is likely to scale poorly and be prohibitively expensive, since extensive inter-MSU communication would be needed to coordinate accesses to the shared memory system. In contrast, a single, centralized, external SMC should perform well for a moderate number of processors. A third, hybrid approach places the SBUs on-chip while the centralized access-order/issue logic remains external.

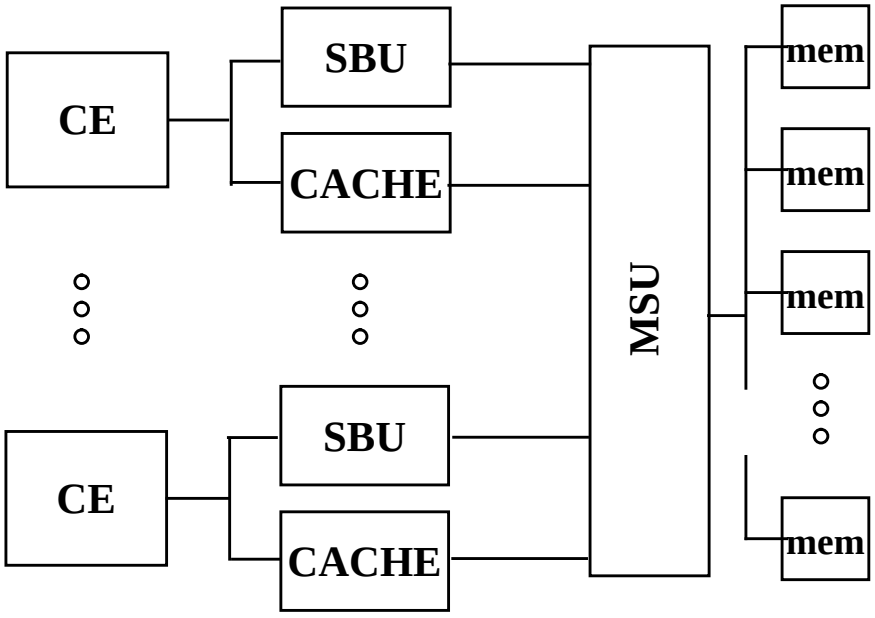



**Figure 2 Symmetric Multiprocessor SMC Organization**

In the SMP SMC system in Figure 2, all computational elements are interfaced to memory through a centralized MSU. The architecture is essentially that of the uniprocessor SMC, but with more than one CE and a corresponding SBU for each. Note that since cache

placement does not affect the SMC, the system could consist of a single cache for all CEs or separate caches for each. Figure 2 depicts separate caches to emphasize the fact that the SBUs and cache reside at the same level of the memory hierarchy.

Due to both the high communication requirements for a fully distributed approach and the limitations on the number of processors that may share a centralized resource, we do not expect SMP SMC systems to scale to large numbers of processors. This paper focuses on the performance of SMP systems with two to eight computational elements.

## 3. Simulation Environment

In order to validate the SMC concept for shared-memory multiprocessor systems, we have simulated a wide range of SMC configurations and benchmarks, varying:

- FIFO depth,
- dynamic order/issue policy,
- number of memory banks,
- DRAM speed,
- benchmark algorithm,
- vector length, stride, and alignment with respect to memory banks, and
- number of computational elements.

The simulations discussed here use unit-stride vectors aligned to have no DRAM pages in common. All memories modeled consist of interleaved banks of page-mode DRAMs, where each page is 4K bytes (512 double words), and the DRAM page-miss cycle time is four times that of a DRAM page-hit. These memory system parameters are representative of current technology. SMC initialization requires two writes to memory-mapped registers

for each stream. Since this small overhead does not significantly affect our results, it is not included here.

Arithmetic and control are assumed never to be a computational bottleneck, thus we model the processor as a generator of load and store requests only. This places maximum stress on the memory system by assuming a computation rate that out-paces the memory's ability to transfer data. Scalar and instruction accesses are assumed to hit in the cache for the same reason.

We chose vector lengths of 10,000 and 80,000 elements for our experiments. Here "vector length" refers to the amount of data processed by the entire parallel computation, not just by one CE. The 10,000-element vectors facilitate comparisons between SMP and uniprocessor systems, since this is one of the vector lengths used in the uniprocessor SMC studies [McK93a, McK93c]. These vectors are long enough that SMC startup transients become insignificant in most cases, but as the number of CEs increases, the amount of data processed by each CE decreases, and startup effects become more evident under certain parallelization techniques. We present 8-CE SMC simulation results for 80,000-element vectors in order to demonstrate the performance of such systems under larger workloads.

We assume that the system is balanced so that the bandwidth between the SMC and memory equals that between the computational elements and the SMC. The number of accesses that the MSU can issue in one processor cycle is thus proportional to the number CEs. The order of future accesses is computed concurrently with the initiation of the current set of accesses. A *ready access* refers to an empty position in a read FIFO (that position is ready to be filled with the appropriate data element) or a full position in a write FIFO (the corresponding data element is ready to be written to memory).

Note that if the number of memory banks in the system *exceeds* the depth of the FIFOs in the SBU, more than one bank will take turns servicing a single FIFO position. This can have

some interesting effects on performance, depending on the SMC's dynamic ordering policy and the nature of the computation. For instance, under certain conditions, shallower FIFOs may deliver a greater percentage of peak bandwidth than deeper ones. These performance anomalies are discussed in Section 5.3. Figure 3 shows an example mapping of memory banks to FIFO positions for a stride-one vector when the length of the FIFOs is less than the number of banks.

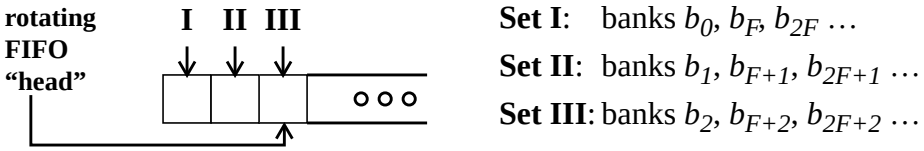



**Figure 3 Mapping of Memory Banks to Positions in a FIFO of Depth *F***

### 3.1 Benchmark Suite

The benchmark suite used is the same as in previous SMC studies [McK93a, McK93c, McK94a, McK94b], and is described in Figure 4. These benchmarks represent access patterns found in real scientific codes, including the inner-loops of blocked algorithms. The suite constitutes a representative subset of all *possible* access patterns for computations involving a small number of vectors. The *hydro* and *tridiag* benchmarks share the same access pattern, thus their results for these simulations are identical, and are presented together.

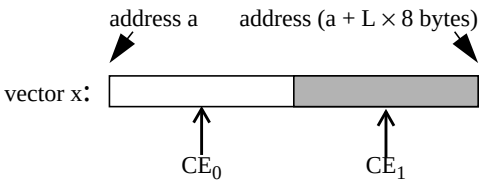
### 3.2 Task Scheduling

The way in which a problem is partitioned for a multiprocessor system can have a marked effect on performance. Three general scheduling techniques are commonly used to parallelize workloads: *prescheduling*, *static scheduling*, and *dynamic scheduling* [Ost89].

| | | |
|---|---|---|
| copy: | $\forall i$ | $y_i \leftarrow x_i$ |
| daxpy: | $\forall i$ | $y_i \leftarrow ax_i + y_i$ |
| hydro: | $\forall i$ | $x_i \leftarrow q + y_i \times (r \times zx_{i+10} + t \times zx_{i+11})$ |
| scale: | $\forall i$ | $x_i \leftarrow ax_i$ |
| swap: | $\forall i$ | $tmp \leftarrow y_i \quad y_i \leftarrow x_i \quad x_i \leftarrow tmp$ |
| tridiag: | $\forall i$ | $x_i \leftarrow z_i \times (y_i - x_{i-1})$ |
| vaxpy: | $\forall i$ | $y_i \leftarrow a_i x_i + y_i$ |

**Figure 4 Benchmark Algorithms**

Prescheduling requires that the programmer divide the workload among the CEs before compiling the program. There is no notion of dynamic load balancing with respect to data size or number of CEs. This type of scheduling is particularly appropriate for applications exhibiting functional parallelism, where each CE performs a different task. Since performance on a single CE is relatively independent of access pattern [McK93a], we model prescheduled computations by running the same benchmark on all CEs. The vector is split into approximately equal-size pieces, and each CE performs the computation on a single piece. Figure 5 depicts this data distribution for a stride-one vector and the corresponding code for the inner loops on a 2-CE system.




CE$_0$'s code:

```
for (i = 0; i < L/2; i++) {
   /* operations on x[i]*/
}
```

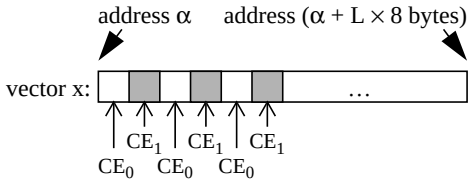
CE$_1$'s code:

```
for (i = L/2; i < L; i++) {
   /* operations on x[i]*/
}
```

**Figure 5 Prescheduling: Data Distribution for 2-CE System**

In static scheduling, tasks are divided among the CEs at runtime, but the partitioning is performed in some predetermined way. Thus a process on a CE must determine which tasks it must do, perform that work, then wait for other processes to finish their tasks. We model static scheduling by distributing loop iterations among the CEs, as in a FORTRAN DOALL

loop. If there are $M$ CEs participating in a computation whose vector stride was originally $S$, this parallelization scheme makes the *effective stride* for the computation $M \times S$ at each CE. Figure 6 illustrates the data distribution and code for this scheme. Note that a static scheduling scheme could break the vector data into hunks, but the SMC simulation results for such a scheme would be identical to those presented for prescheduling.




CE$_0$'s code:
```
for (i = 0; i < L; i += 2) {
   /* operations on x[i]*/
}
```

CE$_1$'s code:
```
for (i = 1; i < L; i += 2) {
   /* operations on x[i]*/
}
```

**Figure 6 Static Scheduling: Data Distribution for a 2-CE System**

In dynamic scheduling, a pool of tasks is maintained. Each CE schedules its own tasks by repeatedly removing a task from the pool and performing it; if the pool is empty, the CEs wait for tasks to appear. Recall that within our simulation framework, arithmetic and control are abstracted out of the computation. Since we are only simulating the inner loops, SMP SMC performance for dynamic scheduling is similar to either prescheduling or static scheduling, depending on how the work is apportioned into tasks. We therefore omit separate simulation results for this scheduling technique.

### 3.3 Ordering Policy

The overwhelming similarity of the performance curves presented in the uniprocessor SMC studies [McK93a, McK93c, McK94a, McK94b] indicates that neither the ordering strategy nor the processor's access pattern has a large effect on the MSU's ability to optimize bandwidth. For moderately long vectors whose stride is relatively prime to the number of memory banks, the SMC consistently delivers nearly the *full* system bandwidth.

In SMP SMC systems, however, there are more factors that can potentially affect performance, thus different scheduling techniques and vector alignments may benefit from different ordering algorithms. For instance, the scheduling technique may:

- change the effective vector stride on any CE (as in static scheduling), and
- affect the "working set" of DRAM pages that are needed during a portion of the computation (with static scheduling, all CEs are likely to be using the same set of DRAM pages, whereas with prescheduling, different CEs are likely to be working on different sets of pages).

The SMC attempts to exploit the underlying memory architecture to issue accesses in an order that optimizes bandwidth. For any memory system composed of interleaved banks of DRAM components, there are at least two facets to this endeavor: taking advantage of available concurrency among the interleaved banks, and taking advantage of device characteristics. At each decision point (each available memory bus cycle), the SMC must decide how best to achieve these goals. The design space of access-order/issue algorithms can be divided into two subspaces: algorithms that first choose a bank (*Bank-Centric* schemes), and algorithms that first choose a FIFO (*FIFO-Centric* schemes).

In order to select the "best" FIFO or bank to use next, an access ordering scheme must either consider all possibilities in parallel, or it must impose some ordering on the resources (FIFOs or banks) so that it can examine them sequentially. Our simulations assume that not all possibilities can be evaluated at once. We therefore model SMC systems using several resource-ordering variations, in order to determine their effects on performance.

For instance, the order in which the FIFOs are considered for service can affect delivered bandwidth. We investigate two different ways in which the MSU selects the next FIFO to service: by examining the FIFOS in sequential round-robin order by processor (all of $CE_0$'s FIFOs are considered before any of $CE_1$'s), and by examining the FIFOs in an interleaved, round-robin order (in which the MSU first considers $FIFO_0$ for $CE_0$, then $FIFO_0$ for $CE_1$, etc., before considering $FIFO_1$ for $CE_0$).

### 3.3.1 The Bank-Centric Approach

In any Bank-Centric ordering policy, the MSU's task can be broken into two subtasks: selecting the banks to which accesses will be initiated, and then deciding which accesses from which FIFOs will be issued to those banks.

*Bank Selection*

We consider two strategies for making the bank selection: *Exhaustive Round-Robin Selection* and *Token Round-Robin Selection*. In the Exhaustive Round-Robin (or just *Exhaustive*) selection scheme, the MSU considers each bank in turn until it has initiated as many accesses as it can, or it has considered all banks. This strategy starts its search by considering the next bank after the last one to which the MSU initiated an access.

With Token Round-Robin selection (*Token*), the MSU only considers a subset of the banks at each decision point, attempting to issue accesses to the idle ones. We examine two different ways of partitioning the banks into subsets. If the MSU can issue up to $N$ accesses at a time, the first algorithm considers the next set of $N$ banks in sequence. Thus the first set contains banks $\{b_0, \ldots, b_{N-1}\}$, the second contains $\{b_N, \ldots, b_{2N-1}\}$, and so forth. In the second variation, a set contains all banks whose indices are congruent modulo the number of CEs: $\{b_0, b_N, b_{2N}, \ldots\}$, etc.

*FIFO Selection*

Once the MSU has selected a set of banks, it must then decide which accesses to issue. We examine two related schemes for choosing the FIFO to service. We refer to the first FIFO-selection policy as simply the *Bank-Centric* algorithm, or *BC*. For a selected memory bank, $b_i$, the algorithm examines the FIFOs in order, beginning with last FIFO for which an access to $b_i$ was initiated. If the MSU finds a ready access that hits $b_i$'s current DRAM page, it issues that access. If no ready accesses for the bank hit the current DRAM page, then an access is issued for the FIFO requiring the most service from bank $b_i$.

The second FIFO-selection algorithm is a more sophisticated variant of the first. Consider the case where no ready accesses hit the current DRAM page. Instead of initiating an access for the FIFO requiring the most service from the current bank, the MSU issues an access *only* if a FIFO meets a certain threshold-of-service criterion. In this case, the portion of a read FIFO for which the current memory bank is responsible must be at least half empty, or the corresponding portion of a write FIFO must be at least half full. This ensures that there will be several fast accesses over which to amortize the cost of switching the DRAM page. We refer to this scheme as the *Threshold Bank-Centric* algorithm, or *TBC.*

### 3.3.2 The FIFO-Centric Approach

The second, general class of access-ordering schemes contains those that first choose a FIFO to service, and then issue accesses from that FIFO to their corresponding banks as appropriate. We investigate a very simple *FIFO-Centric* algorithm, or *FC*: the SMC looks at each FIFO in turn, issuing accesses for the same FIFO stream while:

1) not all elements of the stream have been accessed, and
2) there is room in the FIFO for another read operand, or another write operand is present in the FIFO.

If the current FIFO contains no ready accesses to an idle bank, no access is initiated.

### 3.3.3 Algorithms Simulated

There are many possible means of choosing which banks to access, which FIFOs to service, and in what order to consider each of these resources in making these decisions, and these elements can be combined in myriad ways. Here we focus on five strategies that generally perform well and are representative examples from the design space of dynamic ordering policies:

1) Exhaustive Round-Robin Bank-Centric selection with sequential bank sets,

2) Token Round-Robin Bank-Centric selection with sequential bank sets,

3) Token Round-Robin Bank-Centric selection with modular bank sets,

4) Token Round-Robin Threshold Bank-Centric selection with sequential bank sets, and

5) FIFO-Centric Selection

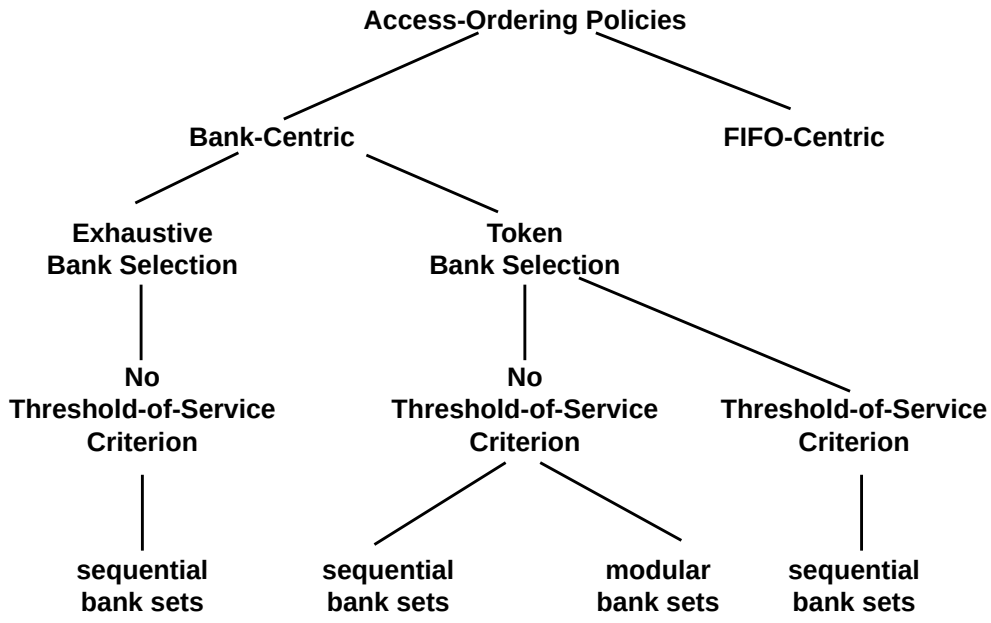



**Figure 7 The Five Ordering Policies**

We expect Token selection to perform about the same as exhaustive selection, but the former should be less expensive to implement. We investigate two types of Token selection — one using sequential bank sets and one using modular band sets — in order to determine what effects the bank-ordering scheme has on performance. We also look at Token selection with a threshold-of-service requirement to determine whether implementing a threshold criterion improves performance, and if so, by how much. Finally, we compare the performance of the Bank-Centric approaches to that of our simple, FIFO-Centric policy. FC

is the most economical policy to implement, but we expect that it will not perform as well as the more sophisticated BC policies for some system configurations and workloads. The relationships between the elements of these ordering strategies can be represented as a tree in which the path to each leaf designates a particular policy, as in Figure 7.[1]

## 4. Performance Factors

The percentage of peak bandwidth delivered is ultimately determined by the MSU's ability to exploit both fast accesses (in the form of DRAM page hits) and the memory system's concurrency. The MSU's effectiveness can be influenced by several factors, including:

- data distribution
- FIFO depth, and
- workload distribution.

These contribute in varying degrees to SMP SMC performance, thus we first take a closer look at them in order to better interpret the results presented in Section 5 and the Appendix.

### 4.1 Data Placement

SMC performance is dramatically affected by whether the working sets of DRAM pages needed by different CEs overlap during the course of the computation. If they do overlap, the set of FIFOs using data from a page will be larger. With more buffer space devoted to operands from that page, more (fast) accesses can be issued to it in succession.

Each DRAM page holds 512 double-word vector elements. Thus on an 8-way interleaved memory, for instance, we incur an initial page miss on each bank, but the computation does not cross page boundaries until $512 \times 8 = 4096$ elements of a given vector have been accessed. On a 16-bank system, the vectors cross DRAM page boundaries at element 8192; on a 32-bank system, at element 16,384; and so on. Figure 8 illustrates the layout of a vector

1. In the uniprocessor SMC study, FC is called *A1*, Token BC is *T1*, Token TBC is *T2*, and Exhaustive BC is *R1*. [McK93a].

with respect to DRAM pages for cases where the page size times the interleaving factor is slightly less than the amount of data to be processed at each of $M$ CEs.

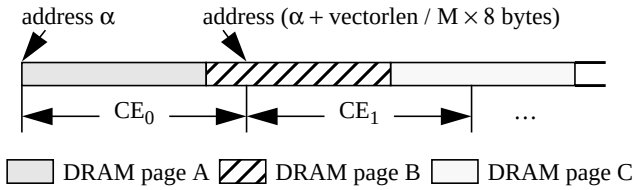



**Figure 8 Example Vector Layout in Memory**

*Prescheduling*

On a 2-CE system with 8 banks, prescheduling divides a 10,000-element vector so that each CE processes approximately 5000 elements, thus the streams for the two computational elements never share pages during the computation. The data layout for each bank is pictured in Figure 9. This figure presents much the same information as in Figure 8, except that the vector chunks for each CE have been arranged vertically to emphasize the portions of data that are being processed in parallel by the different CEs.

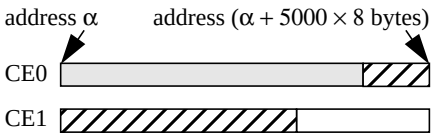



**Figure 9 Distribution of 10,000-Element Vector for 8 Banks and 2 CEs**

Figure 10 shows the distribution of the same 10,000-element vector on a 4-CE system with 8 banks; the pattern of DRAM page-sharing between CE 0 and CE 1 is essentially the same as for a 2-CE, 16-bank system (but in that case each CE would process 5000 elements). CEs 0 and 1 share DRAM pages for almost two-thirds of the computation, and CEs 3 and 4 share for the initial one-third. At the end, CEs 2 and 3 will be on the same pages.

On a 4-CE system with 16 banks, all CEs share the same pages for about one-third of the computation, with three CEs sharing throughout. On a 32-bank, 4-CE system the

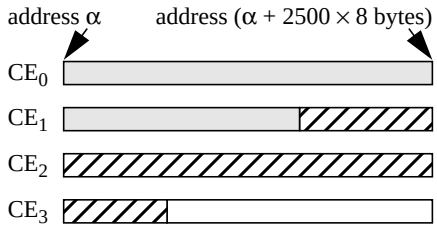



**Figure 10 Distribution of 10,000-Element Vector for 8 Banks and 4 CEs**

computation never crosses a page boundary. This high degree of page-sharing among CEs maximizes the MSU's ability to issue fast accesses.

When we use prescheduling to parallelize a computation on 80,000-element vectors, no page-sharing among CEs is possible for the modest-size SMP systems we investigate here. For an 8-CE system, the data is divided so that each CE processes 10,000 elements. Thus each CE crosses at least two DRAM page boundaries during its computation. This data layout, pictured in Figure 11, causes the MSU to switch DRAM pages frequently, which decreases effective bandwidth.

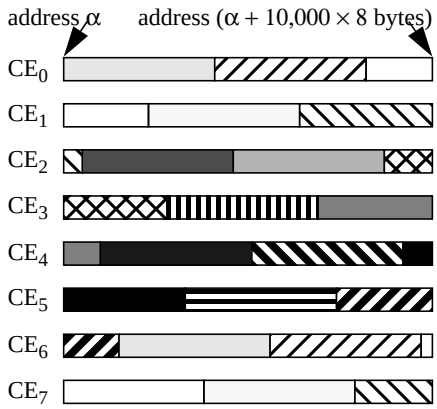



**Figure 11 Distribution of 80,000-Element Vector for 8 Banks and 8 CEs**

*Static Scheduling*

For static task scheduling, each of $M$ CEs performs every $M$th iteration of the loop being parallelized. Thus all CEs access the same set of DRAM pages during any phase of the computation, resulting in fewer page misses and higher bandwidth.

### 4.2 FIFO depth

The second factor affecting SMC performance is FIFO depth. The effect of using deeper FIFOs is similar to that for increasing DRAM page-sharing among the CEs: deeper FIFOs provide more buffer space devoted to operands from a given page, enabling the MSU to amortize DRAM page-miss overheads over a greater number of fast accesses. This has an interesting effect on SMC behavior at the beginning of a computation.

Recall that under our five access-ordering policies, the MSU attempts to perform as much work as possible for a given FIFO before moving on to another. Thus the first operand of a stream will not be fetched until its memory bank has performed as many fast accesses as it can for the FIFOs preceding it. As FIFO depth increases, the CEs must wait longer for all the operands of the first iteration of the computation. If the amount of data to be processed by each CE is small relative to the FIFO depth, there will not be sufficiently many accesses in the computation over which to amortize these startup costs. This results in lower performance. A detailed analysis of this phenomenon is presented elsewhere.

### 4.3 Workload Distribution

Workload distribution is the third factor influencing SMC performance. Data distribution and FIFO depth can interact to create an uneven distribution of the workload with respect to time. Depending on when a CE starts its computation and on the pattern of DRAM page-sharing among the CEs, some CEs may finish earlier than others. For instance, computational elements with a higher degree of DRAM page-sharing are likely to finish before other CEs. This happens because the MSU accesses their pages more frequently, attempting to perform as many fast accesses as it can before turning to accesses that generate DRAM page-misses. When a CE drops out of the computation, the MSU's pool of potential accesses shrinks. While the last CEs are finishing up at the end of the computation, the MSU may not be able to keep the memory banks busy. As FIFO depth

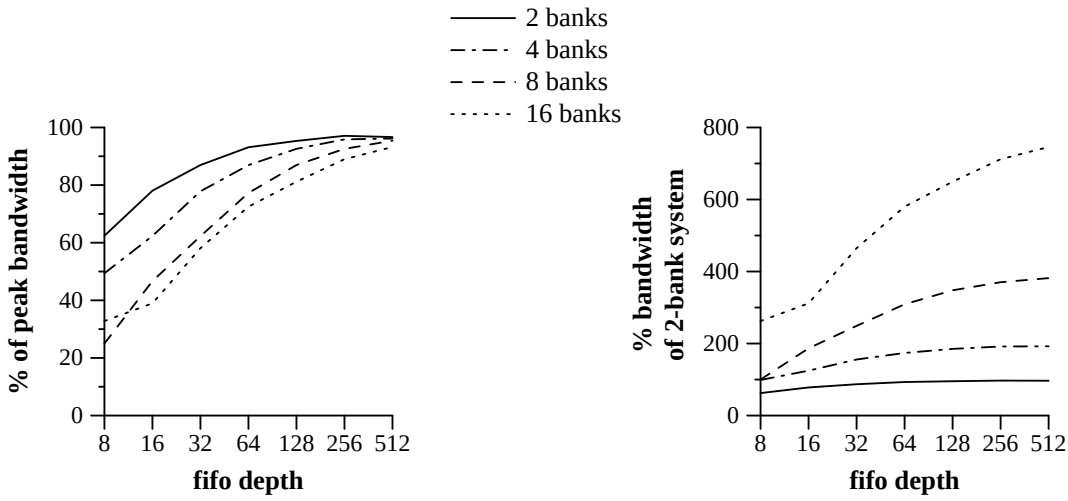
increases, the “faster” CEs tend to finish even earlier, the ending phase becomes longer, and performance suffers even more.

## 5. Results

All results are given as a *percentage of peak bandwidth*, where peak bandwidth represents the performance that would be attained if a memory access could be completed by each CE on every cycle. The performance of each SMP SMC system is presented as a function of FIFO depth and number of memory banks (or available concurrency in the memory system). Unless otherwise stated, all vectors are aligned to DRAM page boundaries, tasks are apportioned such that all vectors (and each CE’s vector chunks, for prescheduled workloads) are aligned to begin in bank $b_0$, and the MSU uses interleaved FIFO ordering. Appendix A gives complete results for all SMC configurations simulated.

The number of memory banks is kept proportional to the number of CEs, thus the curves for an 8-CE system represent performance on a system with four times the number of memory banks as the corresponding curves for a 2-CE system. We keep the peak memory system bandwidth and DRAM page-miss/page-hit cost ratio constant. An 8-bank system therefore has four times the DRAM page-miss latency as a 2-bank system.

Our results demonstrate a common phenomenon: increasing the number of banks often reduces *relative* performance. More banks results in fewer total accesses to each, therefore page-miss costs are amortized over fewer fast accesses. Thus the performance curves for the system with 16 banks represent a smaller percentage of a *much* larger bandwidth when compared to those of a 2-bank system. To illustrate this, Figure 12 depicts SMC performance for the *copy* benchmark on a 2-CE system. Figure 12(a) illustrates performance as a percentage of peak bandwidth for each memory system, and Figure 12(b) depicts the same data as a percentage of peak relative to the total bandwidth of a 2-bank system.




**(a) *normalized bandwidth*** **(b) *scaled bandwidth***

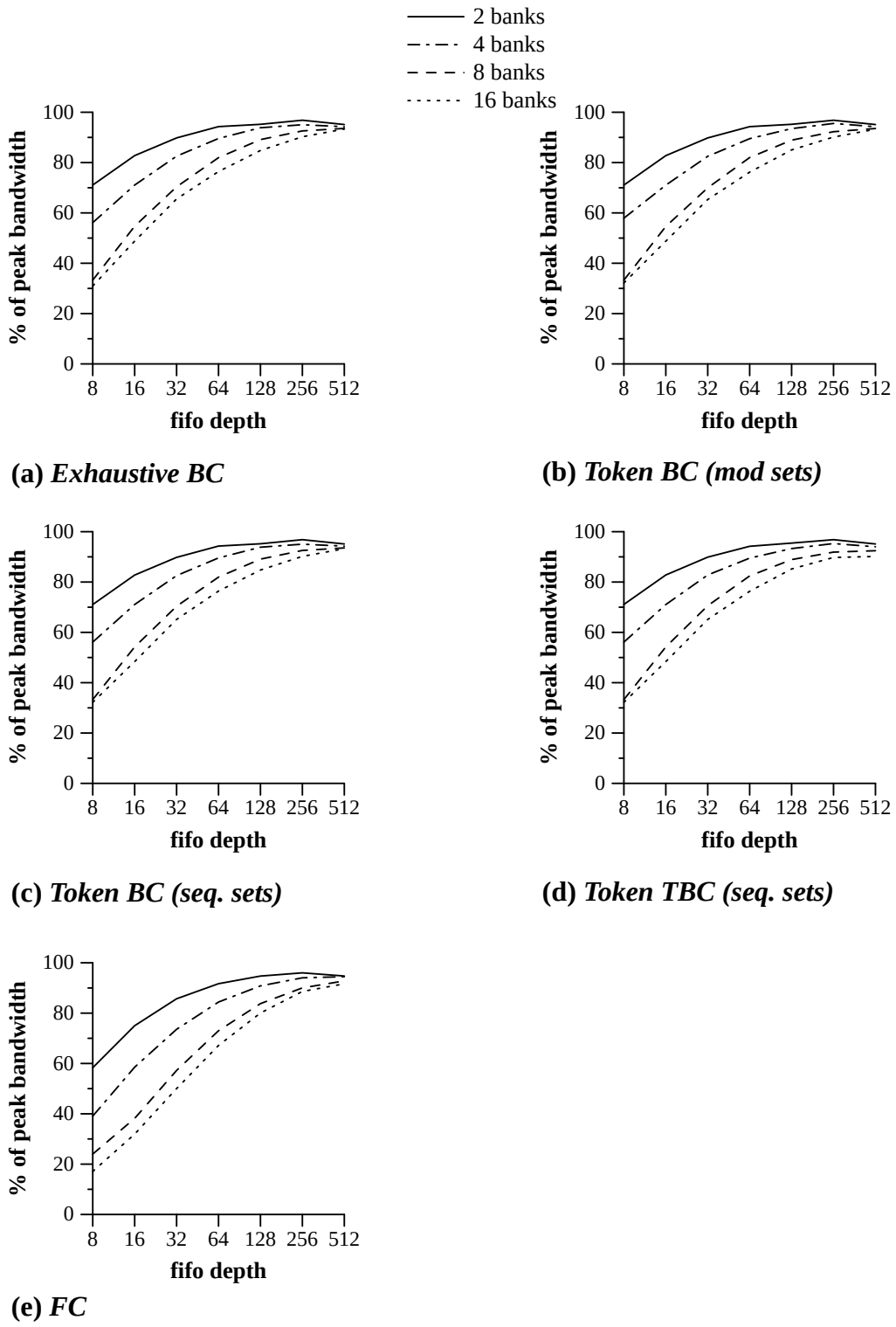
**Figure 12 Prescheduled Token BC *copy* Performance for 2 CEs**

### 5.1 Prescheduling

This section presents SMP SMC performance for prescheduled workloads on systems implementing the ordering policies outlined in Section 3.3. Recall that prescheduling (as described in Section 3.2) essentially breaks the vectors into chunks, assigning each chunk to a different CE to be processed. The effects of changes in relative vector alignment, vector length, or the implementation of an ordering policy (e.g. different FIFO orderings) are fairly independent of the processor's access pattern, thus the graphs presented here focus on a single benchmark, *daxpy.* Complete results for all benchmarks can be found in Appendix A. Like the uniprocessor SMC systems studied [McK93a], SMP SMC performance approaches (and often exceeds) 90% of the peak system bandwidth for sufficiently long vectors and appropriately-sized FIFOs.

Figure 13 through Figure 15 present performance curves for each of our five ordering schemes on SMP SMC systems with 2, 4, and 8 computational elements. Figure 13 illustrates SMC performance for *daxpy* using 10,000-element vectors on systems with 2 CEs. Figure 14 illustrates the same information for SMC systems with 4 CEs, and Figure 15 presents performance curves for 8-CE systems and 80,000-element vectors.

The overwhelming similarity of the curves within each figure (combined with the fact that these results are representative of those for all benchmarks) leads us to conclude that small variations in the dynamic access-ordering policy have little effect on performance. For instance, in most cases Token Bank-Centric ordering (TBC), with its threshold-of-service criterion, performs almost identically to simple Bank-Centric ordering (BC). When their performances differ, TBC delivers a slightly lower percentage of peak.




**(a)** ***Exhaustive BC***

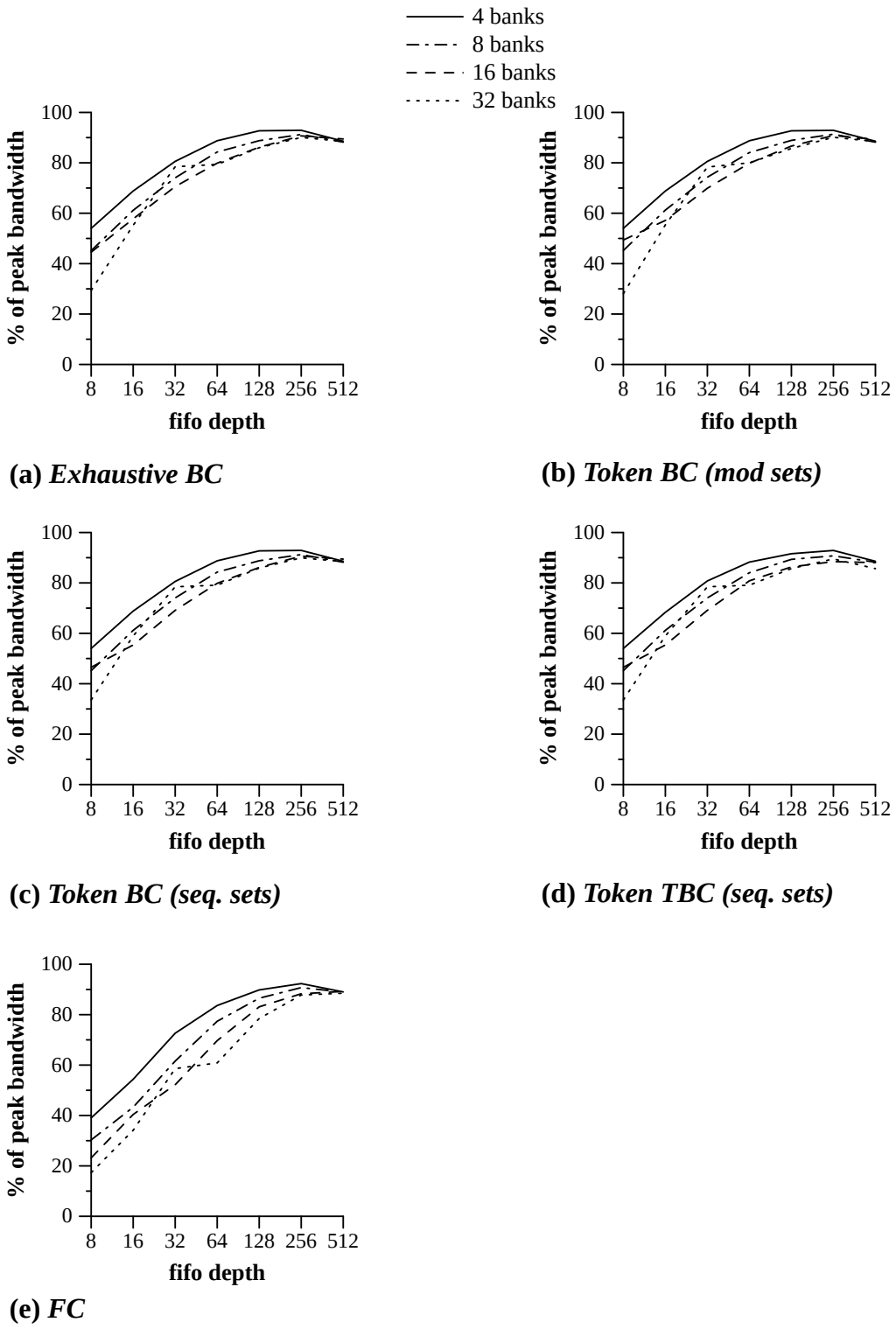
**(b)** ***Token BC (mod sets)***

**(c)** ***Token BC (seq. sets)***

**(d)** ***Token TBC (seq. sets)***

**(e)** ***FC***

**Figure 13 Prescheduled *daxpy* Performance for 2 CEs**




**(a)** ***Exhaustive BC***

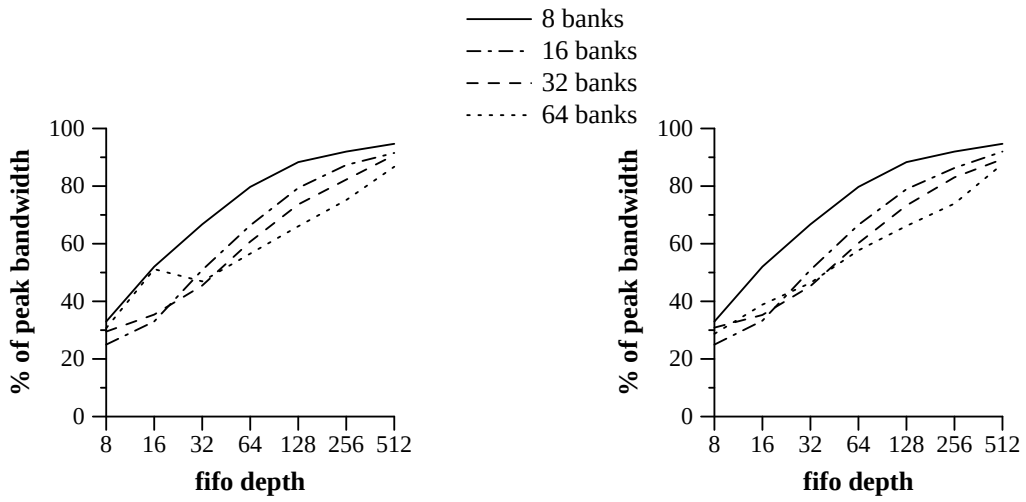
**(b)** ***Token BC (mod sets)***

**(c)** ***Token BC (seq. sets)***

**(d)** ***Token TBC (seq. sets)***

**(e)** ***FC***

**Figure 14 Prescheduled *daxpy* Performance for 4 CEs**




**(a)** ***Exhaustive BC***

**(b)** ***Token BC (mod sets)***

**(c)** ***Token BC (seq. sets)***

**(d)** ***Token TBC (seq. sets)***

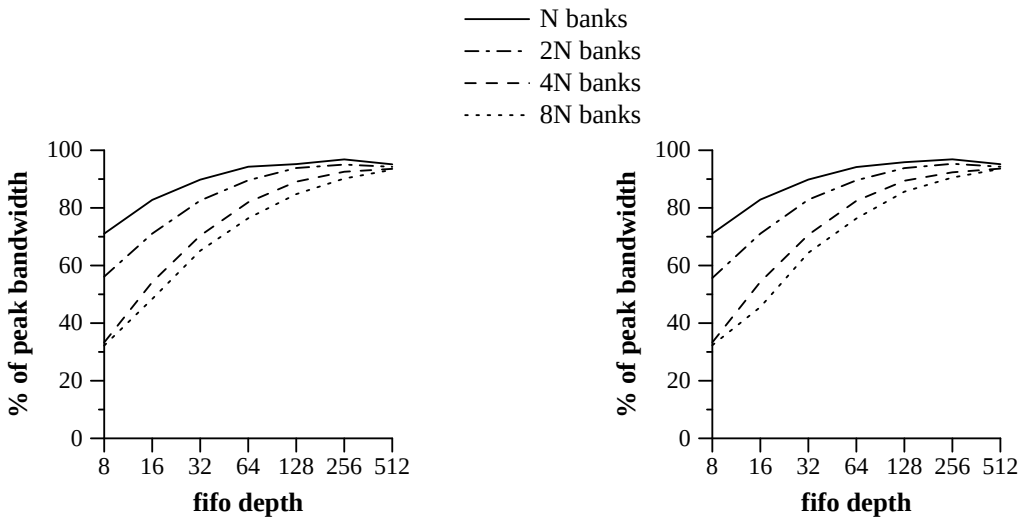
**(e)** ***FC***

**Figure 15 Prescheduled *daxpy* Performance for 8 CEs (Longer Vectors)**

Similarly, Exhaustive bank-selection affords little advantage over either variation of the simpler Token bank selection. The one instance where the exhaustive strategy performs better than the others is for eight CEs, 64 banks, and 16-deep FIFOs. This phenomenon is due more to serendipity than to an inherent superiority of the ordering strategy. Note that this particular system configuration constitutes the one instance exhibiting a noticeable difference in the various schemes' performance: the causes behind this will be examined in Section 5.3.

Note that FIFO-Centric ordering performs slightly worse than Bank-Centric ordering for relatively shallow FIFO depths. The simpler FC scheme concentrates on servicing a single FIFO for as long as possible, thus it cannot take full advantage of DRAM page-sharing among different FIFOs. Nonetheless, for FIFOs of depth 256 or 512, FC's performance is competitive with BC's. Henceforth when we refer to BC access ordering we mean BC using the Token Round Robin variation with sequential bank ordering. This particular scheme is representative of the family of general Bank-Centric schemes: they all perform similarly. Section 5.4 discusses the tradeoffs in implementing BC over FC, or vice versa.

Recall that for the simulations used to generate the results in Figure 12 through Figure 15, all vector chunks were aligned to begin in $b_0$. In order to evaluate the effects of operand alignment on performance, we again simulated our benchmark suite on the various system configurations, this time apportioning the tasks so that the vector data for $CE_i$ begins in bank $b_{i \times (B/N)}$, where $B$ is the number of banks and $N$ is the number of CEs. Figure 16 illustrates performance for *daxpy* using BC ordering with both operand alignments. These results indicate that differences in alignment have little effect on performance for this ordering strategy. The only significant difference occurs for the 4-CE systems with 32 banks and FIFOs of depth 32. In this case the SMC delivers 78.5% of peak bandwidth when the operands are aligned to a single bank, as opposed to 72.3% when they are staggered. This effect is due to bank concurrency, and is discussed in Section 5.3.




**(a)** ***2 CEs, aligned*** **(b)** ***2 CEs, staggered***

**(c)** ***4 CEs, aligned*** **(d)** ***4 CEs, staggered***

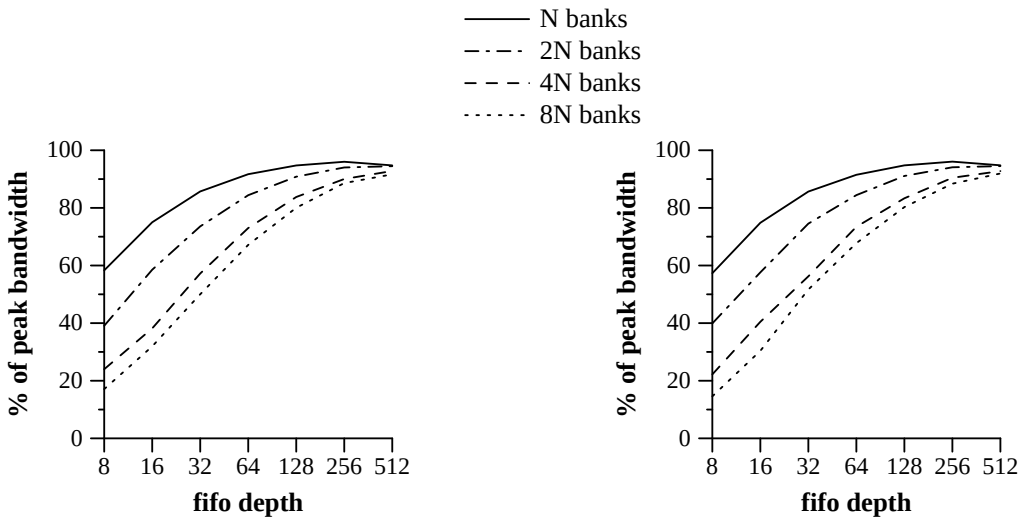
**(e)** ***8 CEs, aligned (long vectors)*** **(f)** ***8 CEs, staggered (long vectors)***

**Figure 16 Effects of Vector Alignment on Prescheduled BC Performance**

SMC systems using FIFO-Centric ordering appear to be more sensitive to differences in operand alignment than those using Bank-Centric ordering. Figure 17 illustrates the performance differences caused by different vector alignments for SMC systems with 2, 4, and 8 CEs. Performance is relatively immune to differences in vector alignment for the 2-CE systems, but performance differences for the 4-CE and 8-CE systems are more pronounced.

In some cases, the staggered alignment yields better bandwidth than when all vectors are aligned to the same bank. A 4-CE system with 8-deep FIFOs, 32 banks, and the staggered vector alignment in Figure 17(d) delivers 11.2% of peak more than the same system using the alignment of Figure 17(c). The a staggered alignment offers a similar performance advantage at a FIFO depth of 64. Unfortunately, staggered alignment offers no consistent advantage. For example, the 4-CE, 32-bank system with 16-deep FIFOs yields over 15% of peak less when vectors are staggered to begin in different banks. Even for deep FIFOs, performance for the staggered alignment is slightly less on this system. Both vector alignments induce essentially the same pattern of DRAM page-sharing among the CEs and cross the same page boundaries, thus the discrepancies result from variations in bank utilization.

The minor implementation details of the SMC's ordering policy have relatively little effect on performance. Figure 13 through Figure 15 demonstrate that permutations in bank order and details such as a threshold of service have little or no effect on delivered bandwidth. Figure 18 illustrates the effects of two different FIFO orderings on performance. If we let the FIFO number indicate its order within a CE's bank of FIFOs and let the subscript indicate the corresponding CE, then the graphs on the left were generated by examining the FIFOs in the order $\{0_0, 0_1, 0_2, \ldots, 1_0, \ldots\}$. Those on the right were generated by considering them in the order $\{0_0, 1_0, 2_0, \ldots, 0_1, \ldots\}$.

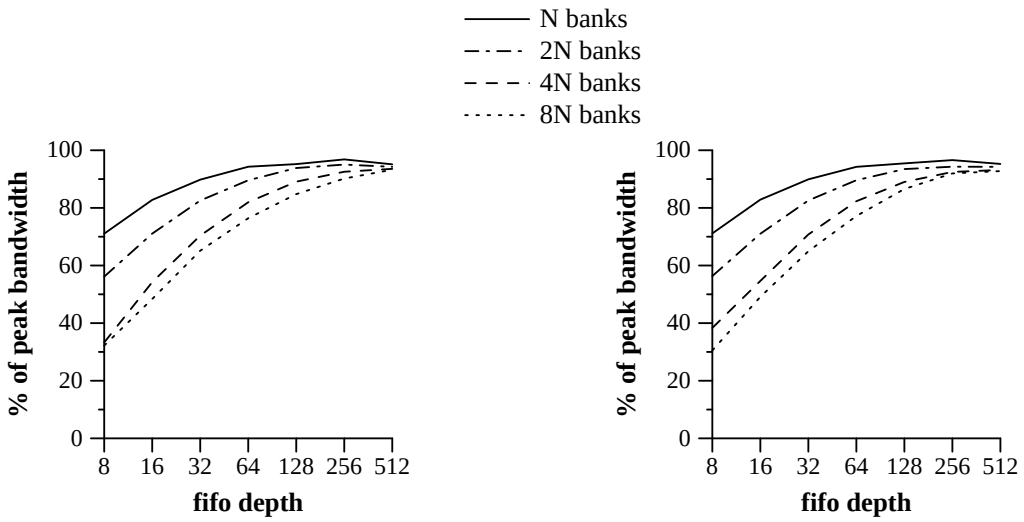



**(a)** ***2 CEs, aligned*** **(b)** ***2 CEs, staggered***

**(c)** ***4 CEs, aligned*** **(d)** ***4 CEs, staggered***

**(e)** ***8 CEs, aligned (long vectors)*** **(f)** ***8 CEs, staggered (long vectors)***

**Figure 17 Effects of Vector Alignment on Prescheduled FC Performance**




**(a)** ***2 CEs, interleaved fifo order***

**(b)** ***2 CEs, sequential fifo order***

**(c)** ***4 CEs, interleaved fifo order***

**(d)** ***4 CEs, sequential fifo order***

**(e)** ***8 CEs, interleaved fifo order***

**(f)** ***8 CEs, sequential fifo order***

**Figure 18 Effects of FIFO Ordering on Prescheduled BC Performance**

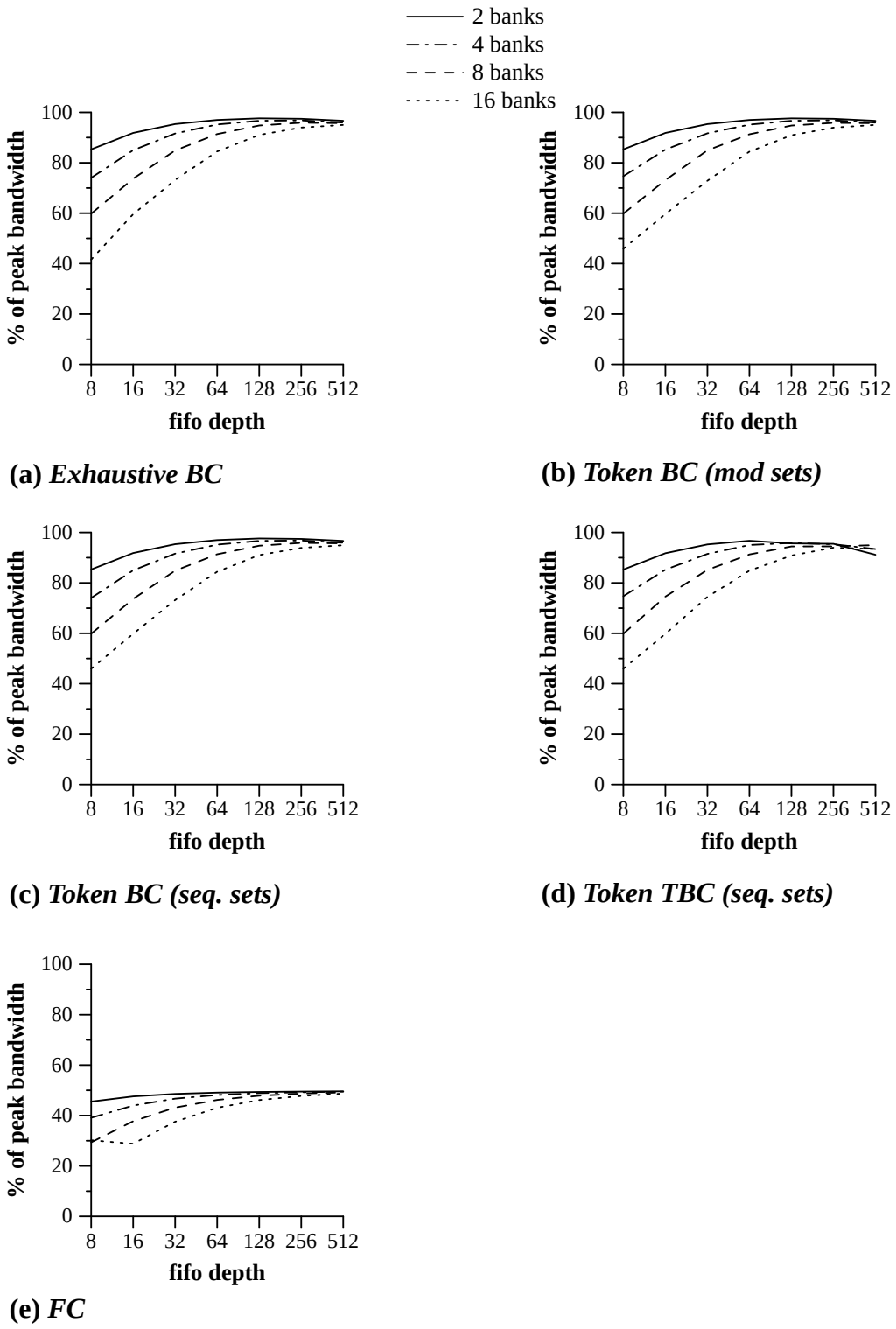
For all benchmarks, differences in the performance of the two algorithm variations are in most cases less than 1% of peak, with neither ordering producing consistently superior results. Larger differences (up to 15% of peak for *scale* using 8-deep FIFOs) occasionally appear for systems in which the number of banks matches or exceeds the FIFO depth. The combination of shallow FIFOs and these particular FIFO orderings can cause poor bank utilization, resulting in significant performance drops.

### 5.2 Static Scheduling

Whereas prescheduling parallelizes a task by breaking a vector into chunks and distributing them among the CEs, static scheduling interleaves loop iterations across the computational elements, thus each of the $M$ CEs participating in a computation would be responsible for every $M$th iteration. Figure 19 through Figure 21 illustrate performance for SMP SMC systems using static scheduling. These systems have 2 to 8 CEs, and all CEs are used in each computation. Since all CEs use the same DRAM pages throughout the computation, the percentages of peak bandwidth delivered by SMP SMC systems using this scheduling technique is almost identical to that for the analogous uniprocessor SMC systems: for long vectors, deep FIFOs, and workloads that allow the MSU to fully exploit bank concurrency, the SMC can consistently deliver almost the full system bandwidth [McK93a].

Figure 19 illustrates the percentages of peak bandwidth attained for the *daxpy* benchmark for 2-CE systems under the five dynamic access-ordering policies on which we've focused. Figure 20 and Figure 21 depict the analogous results for SMC systems with 4 and 8 CEs.

These results demonstrate that SMP SMC performance is insensitive to variations in the Bank-Centric ordering schemes, and is almost constant for a given ratio of CEs to memory banks. For instance, the bandwidth attained by the 8-CE systems with FIFO depths up to 32 differs from that delivered by the analogous 2-CE systems by less than 1% of peak bandwidth. At a FIFO depth of 512, these differences are less than 4.3% of peak.




**(a)** ***Exhaustive BC***

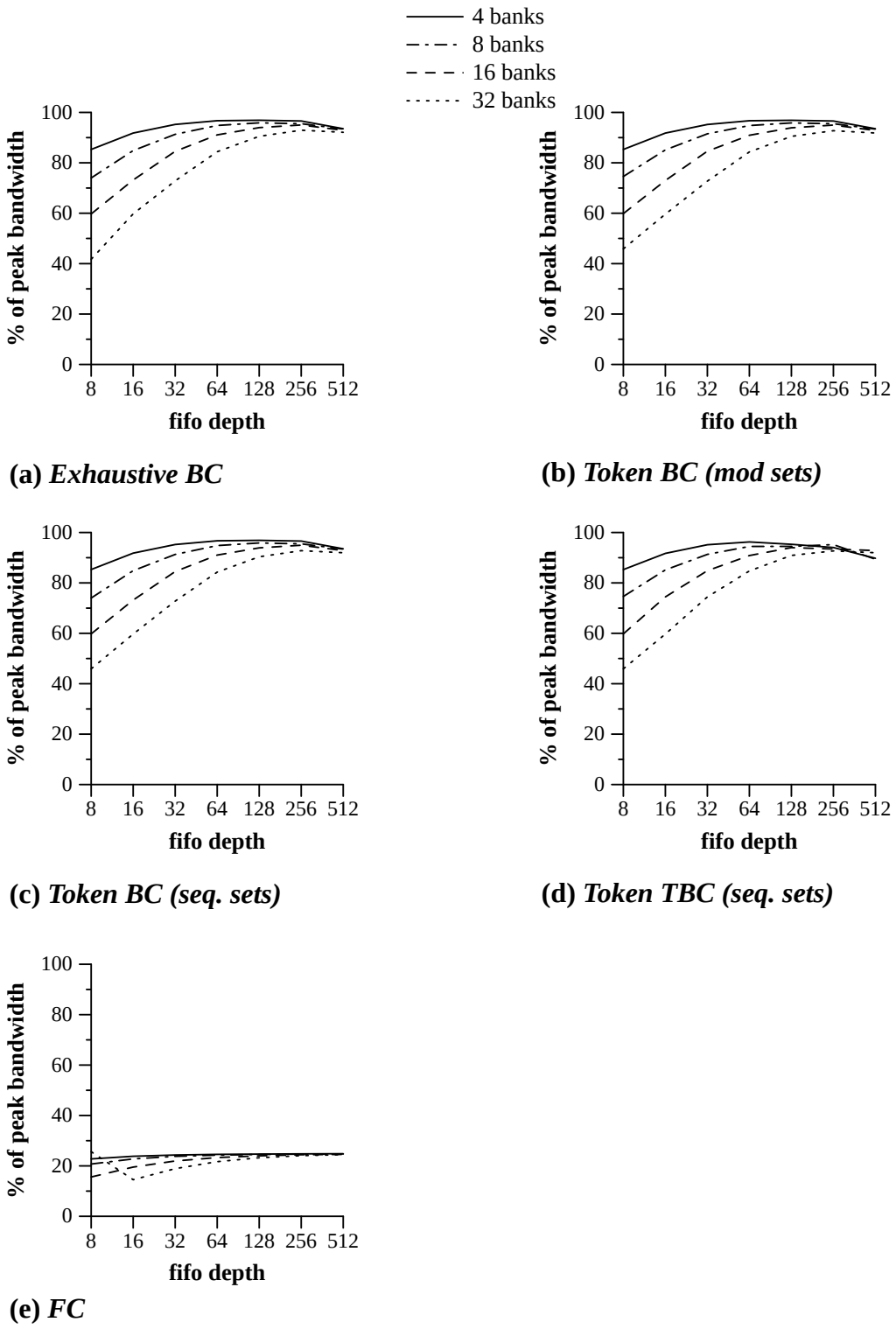
**(b)** ***Token BC (mod sets)***

**(c)** ***Token BC (seq. sets)***

**(d)** ***Token TBC (seq. sets)***

**(e)** ***FC***

**Figure 19 Static *daxpy* Performance for 2 CEs**




**(a)** ***Exhaustive BC***

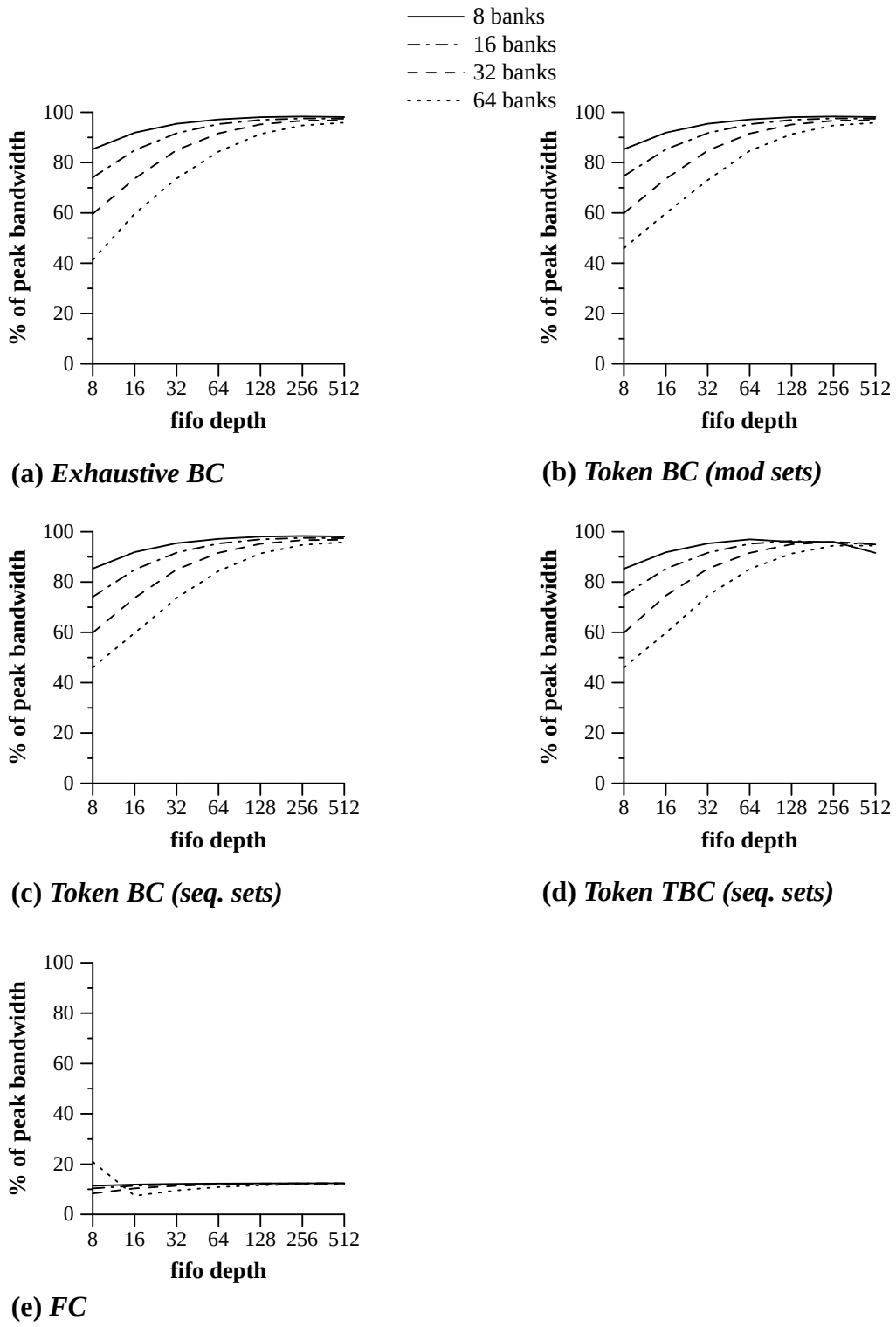
**(b)** ***Token BC (mod sets)***

**(c)** ***Token BC (seq. sets)***

**(d)** ***Token TBC (seq. sets)***

**(e)** ***FC***

**Figure 20 Static *daxpy* Performance for 4 CEs**




**(a)** ***Exhaustive BC***

**(b)** ***Token BC (mod sets)***

**(c)** ***Token BC (seq. sets)***

**(d)** ***Token TBC (seq. sets)***

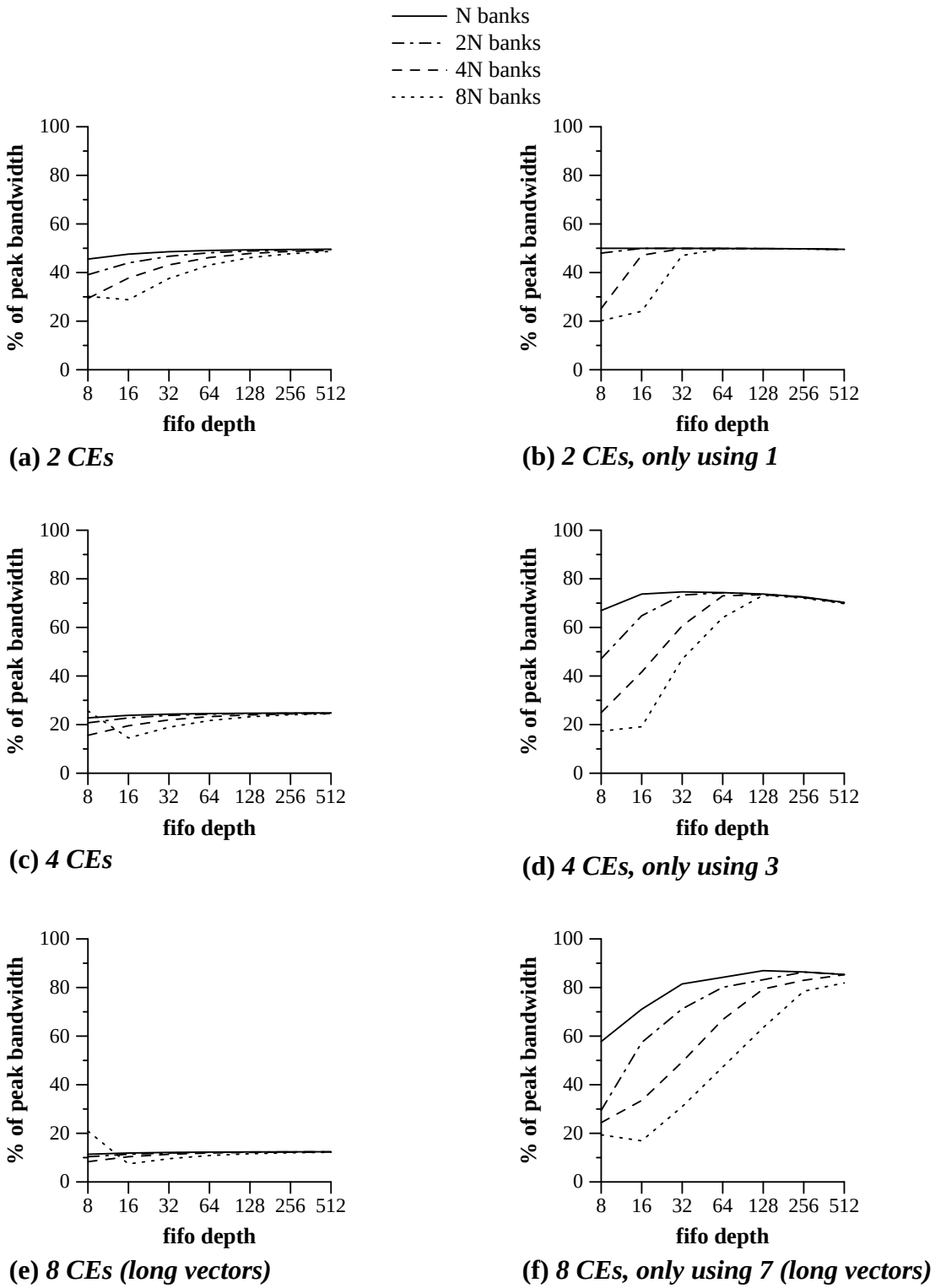
**(e)** ***FC***

**Figure 21 Static *daxpy* Performance for 8 CEs (Longer Vectors)**

In contrast, as the number of CEs increases, attainable bandwidth for the FIFO-Centric scheme is severely limited by lack of bank concurrency. Recall that with static scheduling, the effective stride for each FIFO becomes the natural stride multiplied by $M$, the number of participating CEs, since each CE operates only on every $M$th vector element. The effective stride thus causes each FIFO to use only $1/M$ of the banks used by the natural stride. This means that when $M = N$, an SMC system using FC ordering will probably *not* be able to exploit the full system bandwidth. When all vectors are aligned to begin in the same bank, performance for a computation whose natural stride is relatively prime to the number of banks is generally limited to 50% of peak bandwidth for the 2-CE systems, 25% for the 4-CE system, and 12.5% for the 8-CE systems. Performance for other natural strides will be even lower.

Static scheduling may still be used profitably with FC ordering by using only a *subset* of the CEs, the size of which must be chosen to be relatively prime to the number of memory banks. The effective stride will also be relatively prime, thereby maximizing the MSU's ability to exploit memory system concurrency. Attainable bandwidth becomes limited by the percentage of CEs used, rather than by the percentage of memory banks used. To see this, consider the graphs in Figure 22. The graphs in the left column show *daxpy* performance for SMP SMC systems with FC ordering when all CEs are used. Those on the right indicate performance when one fewer CE is used. Whether or not using fewer CEs yields a net performance gain depends on the total number of CEs and the FIFO depth.

For instance, in Figure 22(a), performance is limited to 50% of peak because the MSU is able to use only one memory bank at a time. Performance is also limited to 50% of peak in Figure 22(b), but for a different reason: here only one computational element is being used. Even though the attainable performance for very deep FIFOs is the same in both cases, performance for shallower FIFOs is not identical: at FIFO depths of 32 to 256, the workloads of Figure 22(b) achieve a greater percentage of peak bandwidth.




**(a)** ***2 CEs*** **(b)** ***2 CEs, only using 1***

**(c)** ***4 CEs*** **(d)** ***4 CEs, only using 3***

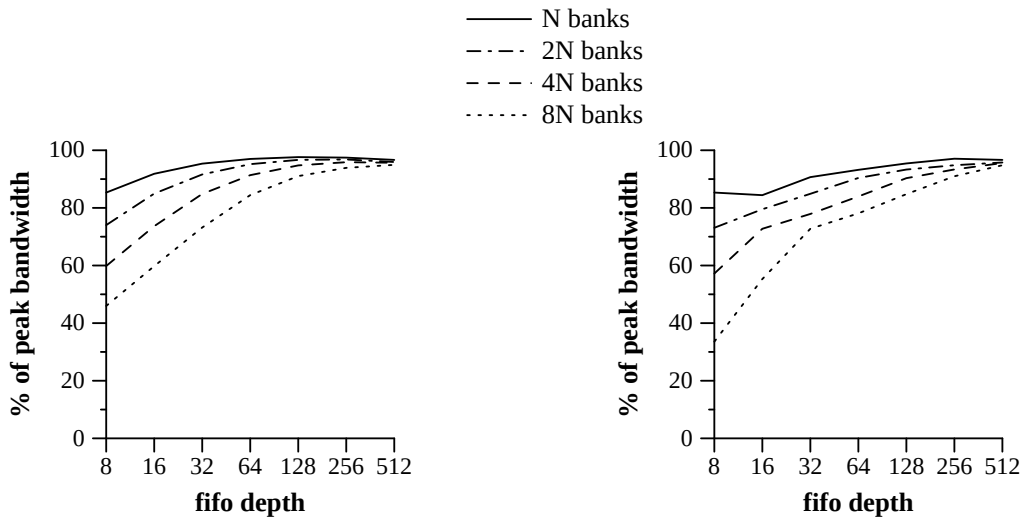
**(e)** ***8 CEs (long vectors)*** **(f)** ***8 CEs, only using 7 (long vectors)***

**Figure 22 Performance Effects of Using Fewer CEs with FC Ordering and Static Scheduling**

Systems with a large number of banks are able to deliver better performance for very shallow FIFOs because the FC ordering mechanism forces the MSU to switch FIFOs often. The phenomenon is evident in the performance curves for systems with $8N$ banks in Figure 22, and will be discussed further in Section 5.3.

In general, for systems with adequate FIFO depth, using one fewer CEs with FC ordering and statically scheduled workloads dramatically improves performance. For example, the 4-CE system with 4 banks in Figure 22(d) delivers 74.6% of peak bandwidth at a FIFO depth of 32, as compared with 24.3% for the analogous system of Figure 22(c). As the number of CEs increases, performance differences become even more dramatic. The 8-CE system with 8 banks in Figure 22(f) delivers 83.2% of peak at a depth of 128 when only 7 CEs are used. In contrast, the same system using all 8 CEs reaches only 12.3% of peak, as depicted in Figure 22(e).

Figure 23 illustrates comparative SMC performance for two different operand alignments. The vectors used to generate the results on the left were all aligned to begin in the same memory bank. For the results on the right, the $i$th vector of the computation was aligned to begin in bank $b_i$. Again, performance is fairly constant for a given ratio of CEs to banks, with all systems delivering almost the full system bandwidth for deep FIFOs. The staggered vector alignment slightly inhibits bank concurrency in systems with relatively shallow FIFOs, hence we see dips in some of the performance curves. For the 2-CE systems, performance differences due to vector alignment are less than 7.5% of peak bandwidth. When FIFOs are only 8-deep, the difference in operand alignment causes a 12.6% of peak drop in performance for the system with 4 CEs and 32 banks. For all other FIFO depths and numbers of banks, the bandwidth delivered for both alignments differs by less than 6.4% of peak. Performance differences are similar for systems with 8 CEs. The differences diminish as FIFO depth increases: at a depth of 512, performances differ by less than 1% of peak for the systems with 2 and 4 CEs, and by less than 3% of peak for the 8-CE systems.




**(a)** ***2 CEs, aligned***

**(b)** ***2 CEs, staggered***

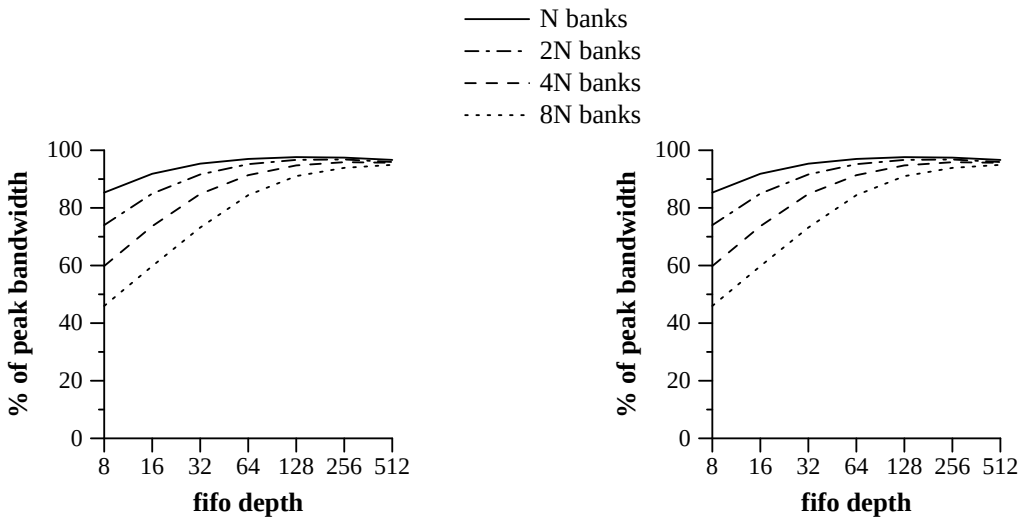
**(c)** ***4 CEs, aligned***

**(d)** ***4 CEs, staggered***

**(e)** ***8 CEs, aligned (long vectors)***

**(f)** ***8 CEs, staggered (long vectors)***

**Figure 23 Effects of Vector Alignment on Static Token BC Performance**




**(a)** ***2 CEs, interleaved fifo order***

**(b)** ***2 CEs, sequential fifo order***

**(c)** ***4 CEs, interleaved fifo order***

**(d)** ***4 CEs, sequential fifo order***

**(e)** ***8 CEs, interleaved fifo order***

**(f)** ***8 CEs, sequential fifo order***

**Figure 24 Effects of FIFO Ordering on Static BC Performance**

Our results indicate that statically scheduled workloads are even less sensitive to changes in operand alignment than prescheduled workloads. Figure 24 depicts the effect of different FIFO orderings on Token BC performance. The graphs on the left illustrate performance when the MSU uses interleaved FIFO ordering; those on the right illustrate performance for sequential ordering. Systems that differ only in the MSU's FIFO ordering deliver essentially identical performances: for all benchmarks, differences are less than 1% of peak bandwidth for FIFO depths of 16 or more, and less than 2% of peak for 8-deep FIFOs.

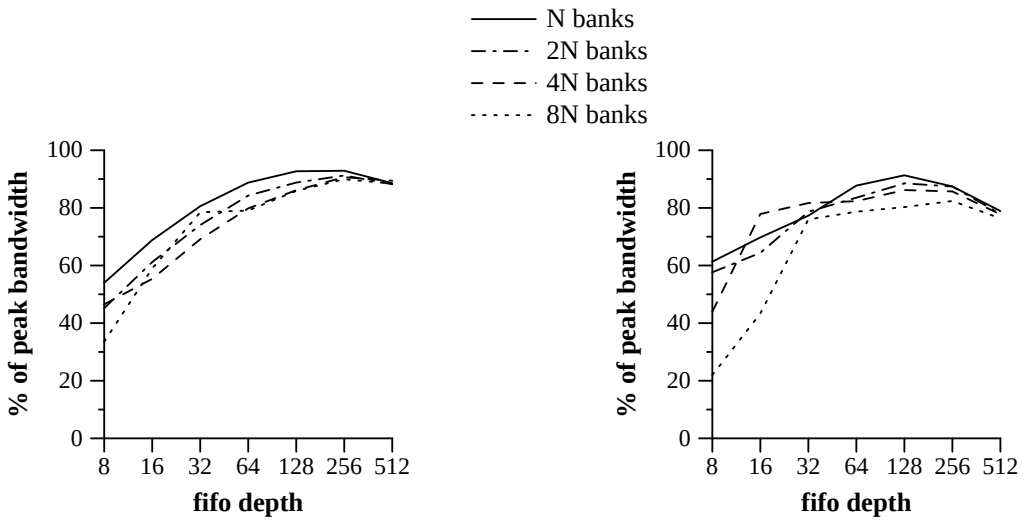
### 5.3 Performance Trends

The performance factors outlined in Section 4 all interact to shape the performance curves presented here. Most curves show bandwidth growing steadily as FIFO depth increases, but several anomalies appear repeatedly throughout many of the graphs. These phenomena can be attributed to startup effects, consequences of the size of the workload on each CE, and general effects due to memory bank utilization and concurrency.

*Tail-Off*

As the number of computational elements increases, the amount of data processed by each element decreases. This contributes to the slight tail-off of the performance curves in Figure 25 as FIFO depth increases. The effect is most pronounced for prescheduled workloads and 8-CE systems using 10,000-element vectors, as in Figure 25(b), Figure 25(f), and Figure 26. Like that encountered for short vectors (100 elements) on uniprocessor SMC systems [McK93a], this phenomenon illustrates the net effect of competing performance factors associated with FIFO depth:

1) The MSU needs sufficiently deep FIFOs to be able to keep the banks busy most of the time and to amortize page-miss costs over a number of page-hits.
2) Deeper FIFOs cause longer startup delays for the CEs, and performance declines when there are not enough accesses over which to amortize startup costs.




**(a)** ***Prescheduled Token BC, 4 CEs***

**(b)** ***Prescheduled Token BC, 8 CEs***

**(c)** ***Static Token BC, 4CEs***

**(d)** ***Static Token BC, 8CEs***

**(e)** ***Prescheduled FC, 4CEs***

**(f)** ***Prescheduled FC, 8 CEs***

**Figure 25 Performance Curve Tail-Off for *daxpy* (10,000-Element Vectors)**

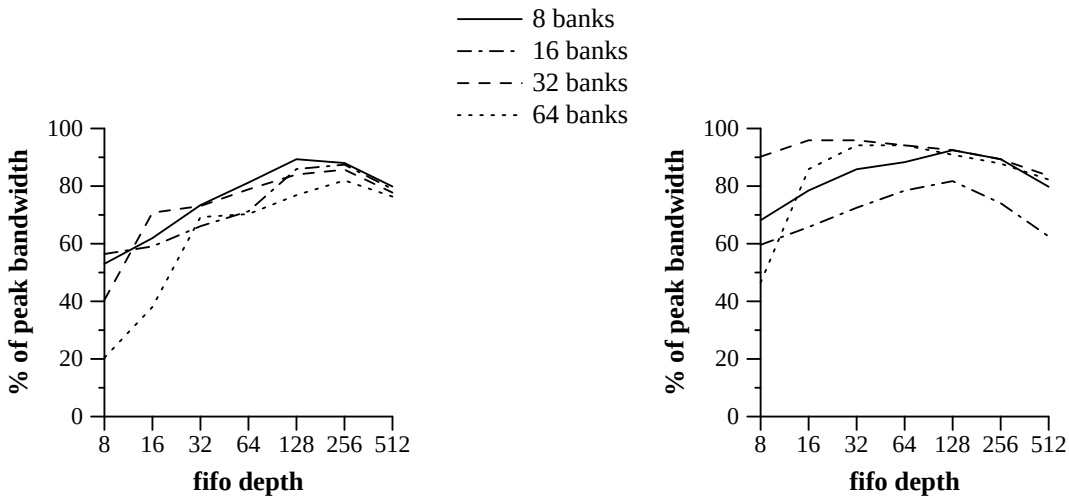
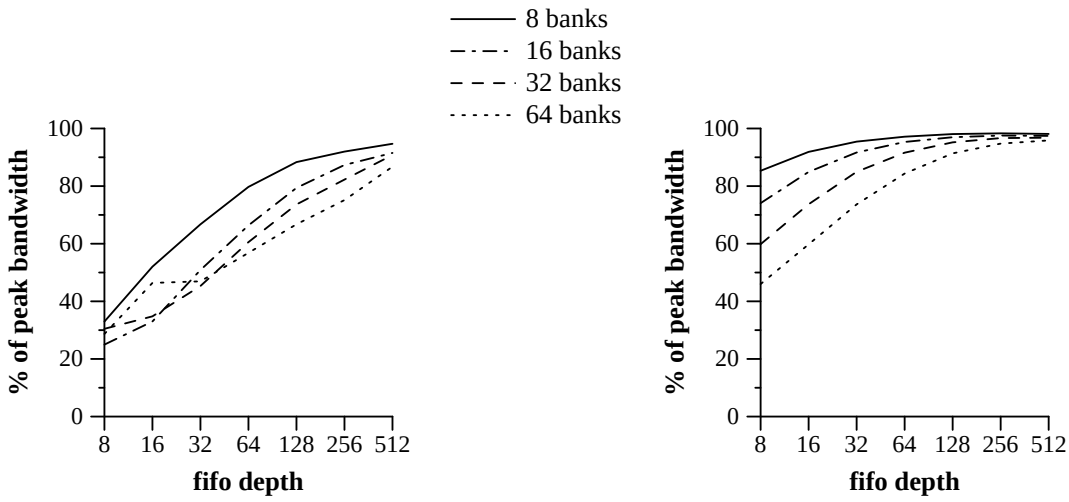



**Figure 26 Prescheduled Token BC Performance for 8 CEs (10,000-Element Vectors)**

Another factor comes into play for prescheduled workloads under BC ordering: shallow FIFOs force the MSU to switch FIFOs fairly often, causing it to service the FIFOs of all CEs relatively evenly. This prevents any one CE from getting too far ahead of the others, resulting in a more even workload for the MSU throughout the course of the computation, This, in turn, can yield better bank utilization.Thus even though shallow FIFOs usually hinder the MSU's ability to optimize bandwidth, they provide an occasional benefit. Unfortunately, the circumstances under which they perform well are hard to predict.

Just as it did in the uniprocessor case, the tail-off effect disappears under larger workloads. This is evident in Figure 27: at a FIFO depth of 512, we have not yet hit the point of diminishing returns.




**(a) *prescheduled*** **(b) *static***

**Figure 27 *daxpy* Token BC Performance for 8 CEs (80,000-Element Vectors)**

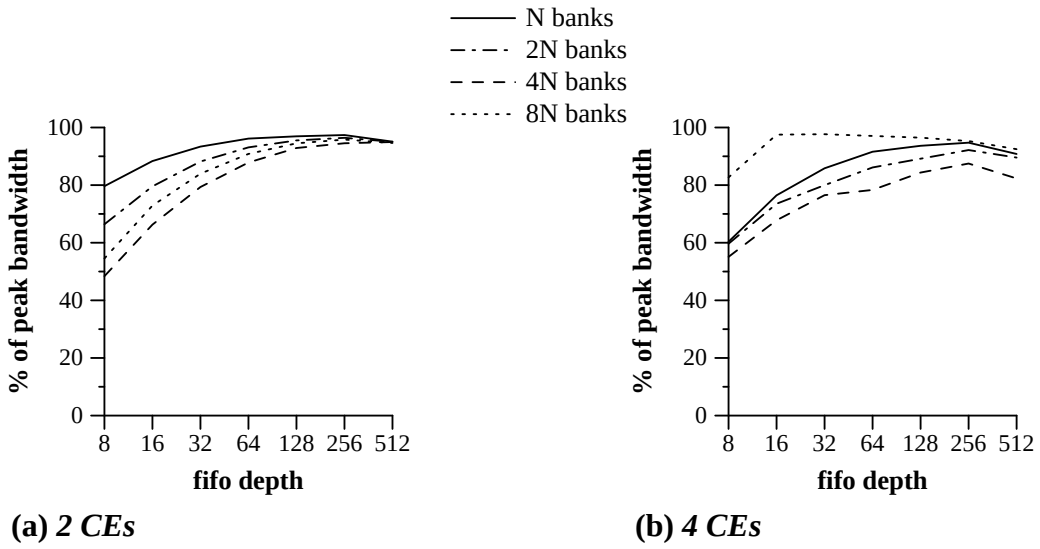
*Higher Performance for More Banks*

Recall that relative bandwidth tends to decrease as the number of memory banks increases. In spite of this, for prescheduled workloads on SMC systems with four and eight CEs and BC access-ordering, systems with a greater number of banks sometimes perform competitively with those with fewer banks. This is due largely to the data partitioning.

For instance, for prescheduled computations on vectors of 10,000 elements, the data is partitioned such that for systems with 32 or 64 banks, all CEs are operating on the same set of DRAM pages. Since the systems with more banks incur fewer page-misses, their raw performance is occasionally as good as or better than that for systems with fewer banks.

The *scale* benchmark is a good example. This benchmark accesses a single vector, thus on systems with 32 or more banks, the computation never crosses a DRAM page boundary. The only page misses are the initial ones at each bank. Given the simplicity of the access pattern and the fact that all CEs are working on the same page, the MSU is able to keep each bank busy most of the time. Thus a system with $N$ CEs and $4N$ or $8N$ banks (and the

extra concurrency they afford) often performs better than one with fewer banks. Figure 28 illustrates this effect for 2-CE and 4-CE systems.




**(a) *2 CEs*** **(b) *4 CEs***

**Figure 28 Prescheduled Token BC Performance for *scale***

An interesting exception to this occurs when FIFO depth is less than the number of banks. In Figure 26(b), the prescheduled *scale* benchmark on an 8-CE, 64-bank system with 8-deep FIFOs is limited to 46.5% of peak: the shallow FIFO depth prevents the MSU from keeping the banks busy. In contrast, each bank in the 32-bank system spends much less time sitting idle, hence the SMC is able to exploit 90.2% of the available system bandwidth. Increasing the FIFO depth increases the available work for each bank at any given time. At depths of 64 or more, systems with 32 and 64 banks perform virtually identically.

*Performance Curve Humps*

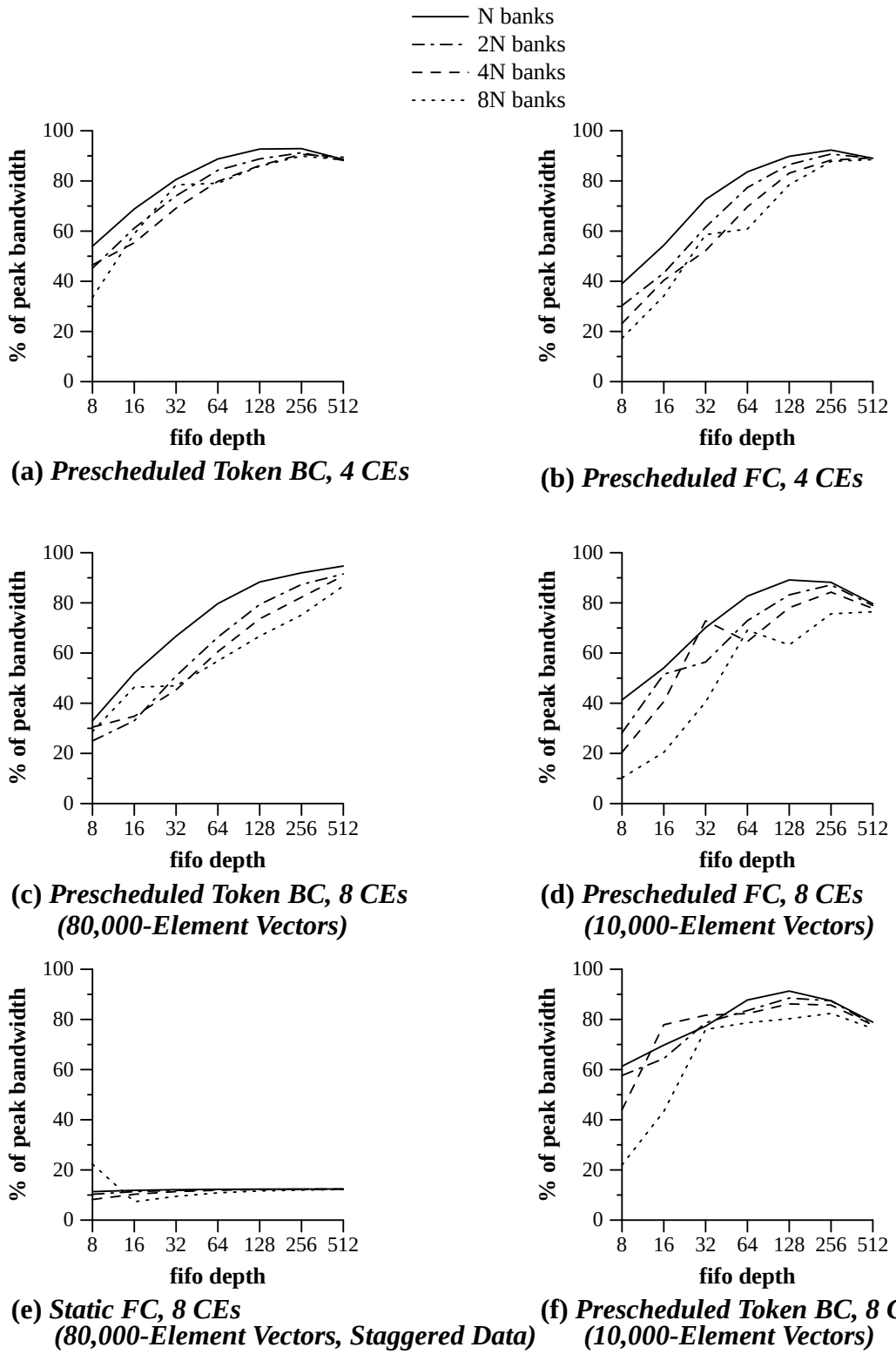
Shallow FIFO depths can sometimes *increase* bank concurrency. For our prescheduled benchmarks, this generally occurs for FIFOs of 16 to 32 elements, and results from the way BC ordering with shallow FIFOs promotes good bank utilization and a more even rate of progress among the CEs. This causes the “humps” in the performance curves of the prescheduled 32 and 64-bank systems in Figure 29. The FIFO depths at which this serendipity occurs depend on the number of streams in the computation, the degree of page-

sharing among the CEs, the number of CEs, the DRAM cycle time, and the number of memory banks.

This effect is less noticeable under a larger workload for the 8-CE systems. The 80,000-element vectors are divided so that each CE processes roughly 10,000 elements. The data layout is such that no CEs share any DRAM pages during any portion of the computation (as in Figure 11). Startup and page-sharing effects are minimized. The MSU must switch between pages more often, and the size of the data set causes the computation to cross more page boundaries, thus the curves in Figure 29(c) are smoother than the corresponding curves for the shorter vectors in Figure 29(f), but performance for shallow FIFOs is much lower.

Another interesting peak occurs in Figure 29(e), which depicts the performance of an 8-CE, 64-bank system using FC ordering for a statically scheduled workload. In general, this phenomenon occurs for systems with a large number of banks and shallow FIFOs. In our simulations, whenever the MSU switches FIFOs, accesses are initiated for the new FIFO while others are still being completed for the old FIFO. If different FIFOs use different subsets of the memory banks, this may yield better bank utilization. Note that in such cases, good performance depends on the FIFO ordering scheme used by the dynamic access-ordering policy: when all vectors are aligned to begin in the same bank, interleaving the FIFOs of all CEs will allow more bank concurrency than servicing the FIFOs of a single CE in sequence.

With the particular data layout of Figure 29(e), the $i$th elements of each vector reside in different banks, thus all FIFOs do not require service from the same set of banks at the same time. The shallow FIFO depth causes the MSU to change FIFOs often. Together, the data alignment and the frequent switching allow the MSU to keep more than $1/N$ of the banks busy at a time. Thus in this case the MSU is able to deliver more than 12.5% of peak bandwidth, in spite of the limitations of FC ordering for non-unit stride vectors.




**(a)** ***Prescheduled Token BC, 4 CEs***

**(b)** ***Prescheduled FC, 4 CEs***

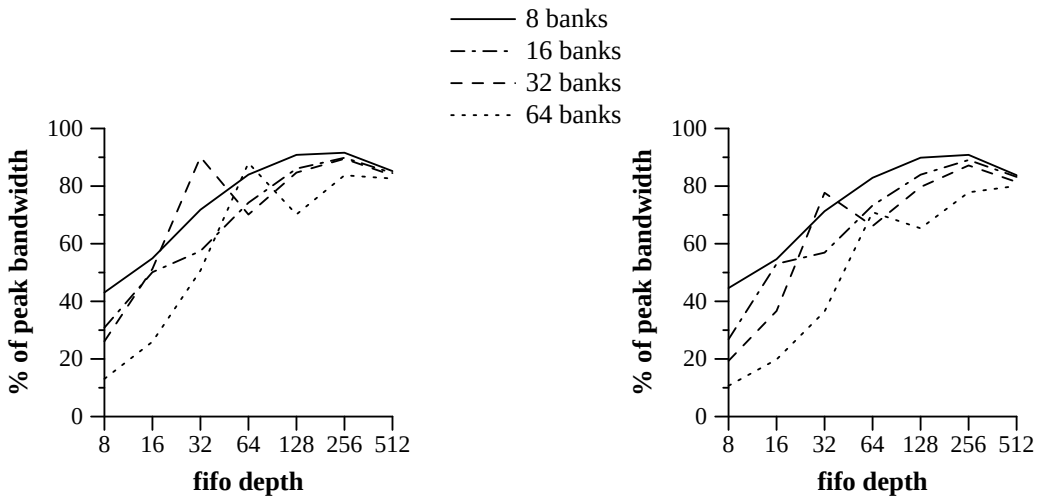
**(c)** ***Prescheduled Token BC, 8 CEs (80,000-Element Vectors)***

**(d)** ***Prescheduled FC, 8 CEs (10,000-Element Vectors)***

**(e)** ***Static FC, 8 CEs (80,000-Element Vectors, Staggered Data)***

**(f)** ***Prescheduled Token BC, 8 CEs (10,000-Element Vectors)***

**Figure 29 Anomalies in Performance Curves for *daxpy***




**(a) *scale*** **(b) *swap***

**Figure 30 Prescheduled FC Performance for 8 CEs (10,000-Element Vectors)**

For SMP SMC systems using prescheduling and FC ordering, these anomalies tend to occur whenever there is a high degree of DRAM page-sharing among the CEs and the FIFO depth equals the number of banks. Systems configured so that FIFO depth matches the interleaving factor allow all banks to work on the same FIFO at once, thereby promoting bank concurrency. The FIFOs are shallow enough that the MSU must switch FIFOs often, thus the CEs proceed at a fairly even pace. More than one CE is using the same set of DRAM pages, so many page-hits are possible. Figure 30 illustrates this effect for *scale* and *swap* with vectors of length 10,000 on 8-CE systems.

### 5.4 Choosing an Access-Ordering Scheme

With shallow FIFOs, performance for our FIFO-Centric ordering policy is consistently lower than that for Bank-Centric ordering. This emphasizes the importance of having sufficiently deep FIFOs for systems with FC ordering.

Of the two families of ordering schemes examined here, FC is easier to implement in hardware, since it requires less information in order to select the MSU's next access. With

deep FIFOs, FC systems end up amortizing DRAM page-miss overheads over a large number of fast accesses, even though the algorithm doesn't explicitly attempt to maximize page hits. For vector strides that are relatively prime to the number of banks, FC is able to successfully exploit the memory system's available concurrency. Under these circumstances, FC's performance is competitive with BC's.

Nonetheless, FC ordering is much more sensitive to changes in vector length and alignment, and FC consistently delivers a lower percentage of peak bandwidth for shallow to medium-depth FIFOs. Moreover, when the vector stride is not relatively prime to the number of memory banks, FC is severely limited in its ability to exploit bank concurrency.

Bank-Centric ordering, on the on the other hand, provides more consistent, robust performance at the cost of slightly more complicated reordering circuitry. The variations to BC ordering that we've investigated here have little impact on performance. No consistent trends are discernible, thus the simplest BC scheme should perform adequately.

Our results indicate that the order in which the MSU considers the FIFOs for service can interact with other performance factors to impact results. The optimal FIFO ordering algorithm would give priority to any FIFOs with accesses to current DRAM pages, and then to the FIFOs that, if not serviced, will cause a CE to stall soonest (either waiting for read data to arrive or for a position in a write FIFO to become available). The two schemes implemented here are simple (and easily implemented) heuristics, neither of which has proved consistently superior to the other.

On SMP SMC systems, Bank-Centric access ordering is the clear implementation choice, for it allows the MSU to exploit locality of DRAM page references across FIFOs for all CEs. If hardware requirements and cost preclude the use of BC ordering, FC ordering may perform adequately, although more care must be taken in parallelizing tasks.

## 6. Recommendations

Our results indicate that appropriately-sized FIFOs and sufficiently many memory banks are required to amortize DRAM page-miss costs and to sustain concurrency in the memory system. However, if these conditions are met, an SMC system can deliver nearly all the bandwidth the memory has to offer.

In general, static scheduling (via interleaving iterations across CEs) yields better memory performance than prescheduling. The advantages to this approach include:

- even workload distribution (the CEs proceed at a more even pace than they generally do using prescheduling),
- DRAM page sharing benefits (in general, all CEs are using the same set of DRAM pages during all phases of the computation).

The exception to this occurs when FC ordering is used for a computation whose effective stride is not relatively prime to the number of memory banks. If all CEs in the system are used for the computation, the attainable bandwidth is limited by the percentage of memory banks used by each FIFO. If instead only a subset of the CEs is used, and if this subset is chosen to be relatively prime to the number of banks, then attainable bandwidth will be limited by the percentage of CEs used.

All our results underscore the importance of using an appropriate FIFO depth for a particular computation: FIFO depth should be settable at runtime. The optimal number of CEs to use depends on the vector length and alignment, parallelization technique, dynamic ordering policy, and parameters of the memory system. For instance, for SMC systems with BC ordering, prescheduled tasks should distributed data so as to promote DRAM page sharing among the CEs whenever possible. In all cases, each CE needs to process enough data to effectively amortize communication costs and SMC startup effects. Equations to determine optimal FIFO depth are presented elsewhere.

## 7. Related Work

There is a large body of research characterizing and evaluating the memory performance of scientific codes. Most of this research focuses on:

a) hiding or tolerating memory latency,

b) decreasing the number of cache misses incurred, or

c) avoiding bank conflicts in an interleaved memory system.

Prefetching and nonblocking caches can be used to overlap memory accesses with computation, or to overlap the latencies of more than one access [Bae91, Cal91, Gup91, Kla91, Mow92, Soh91]. These techniques can improve processor performance, but techniques that simply mask latency do nothing to increase effective bandwidth. Such schemes are still useful, but they will be most effective when combined with complementary technology to exploit memory component capabilities.

Modifying the computation to increase the reuse of cached data can improve performance dramatically [Gal87, Gan87, Car89, Por89, Wol90, Lam91, Tem93]. These techniques will also deliver better performance when integrated with access ordering.

Several schemes for avoiding bank contention, either by address transformations, skewing, or prime memory systems, have been published [Bud71, Gao93, Har87, Har89, Rau91]; we do not discuss them, other than to note that these, too, are complementary to the access ordering.

## 8. Conclusions

As processors become faster, memory bandwidth is rapidly becoming the performance bottleneck in the application of high performance microprocessors to vector-like algorithms. These computations lack the temporal and spatial locality required for caching alone to bridge the performance gap.

Achieving greater bandwidth requires exploiting the characteristics of the *entire* memory hierarchy: it cannot be treated as though it were uniform access-time RAM. This knowledge should guide processor designs and operating system implementations. Mechanisms to take advantage of memory component capabilities should be readily available to the user.

Dynamic access ordering can optimize accesses to exploit the underlying memory architecture. By combining compile-time detection of streams with execution-time selection of the access order and issue, we achieve near-optimal bandwidth for vector-like accesses relatively inexpensively. This complements more traditional cache-based schemes, so that overall effective memory performance need not be a bottleneck. Moreover, dynamic access ordering requires no heroic compiler technology, and is complementary to other common code optimizations.

Here we have reported the basic design of a Stream Memory Controller (SMC) for modest-size symmetric multiprocessor systems, and have analyzed its performance for a wide variety of design parameter values. Our results indicate that for long vectors and deep FIFOs, an SMP SMC system can deliver nearly the full memory system bandwidth.

## Acknowledgments

This work was supported in part by a grant from Intel Supercomputer Division and by NSF grant MIP-9307626.

# Appendix A

Unless otherwise noted, FIFO order is interleaved, Token bank-selection schemes use sequential sets, and vectors are aligned to begin in bank $b_0$. All vectors are 10,000 double-word elements, unless otherwise noted; "longer" vectors are 80,000 elements.

Figure 31 through Figure 66 depict performance for our benchmarks when prescheduling is used to parallelize the tasks. Figure 31 through Figure 34 illustrate the performance of Exhaustive Bank-Centric ordering. Figure 35 through Figure 42 depict results for Token Bank-Centric ordering for both modular and sequential bank sets. Figure 43 through Figure 46 illustrate results for Token Threshold Bank-Centric ordering. Figure 47 through Figure 50 indicate Token BC performance for a different operand alignment. "Staggered alignment" here refers to apportioning the tasks so that the vector data for $CE_i$ begins in bank $b_{i \times (B/N)}$, where $B$ is the number of banks and $N$ is the number of CEs. Figure 51 through Figure 54 present the analogous information for Token TBC ordering. Figure 55 through Figure 58 illustrate the combined effects of staggered operand alignment and sequential FIFO ordering on Token BC performance. Prescheduled FC performance for both operand alignments is illustrated in Figure 59 through Figure 66.

Figure 67 through Figure 100 illustrate SMP SMC performance for our access-ordering policies when static scheduling is used to generate the parallel tasks. For these experiments, "staggered alignment" means that the $i$th vector in the computation begins in bank $b_i$. Figure 67 through Figure 70 show results for Exhaustive Bank-Centric access ordering. Figure 71 through Figure 74 depict the performance of Token Bank-Centric access ordering. Figure 75 through Figure 78 illustrate the analogous data for computations using a staggered vector alignment. Figure 79 through Figure 82 give results for the Token BC algorithm with sequential FIFO ordering when used on computations with staggered vectors. Figure 83 through Figure 90 present data for Token Threshold Bank-Centric ordering for both aligned and staggered vectors. Figure 91 through Figure 96 give results

for FIFO-Centric ordering when all CEs are used, and Figure 97 through Figure 100 show performance for FC ordering when one fewer CE is employed on computations with aligned vector data.

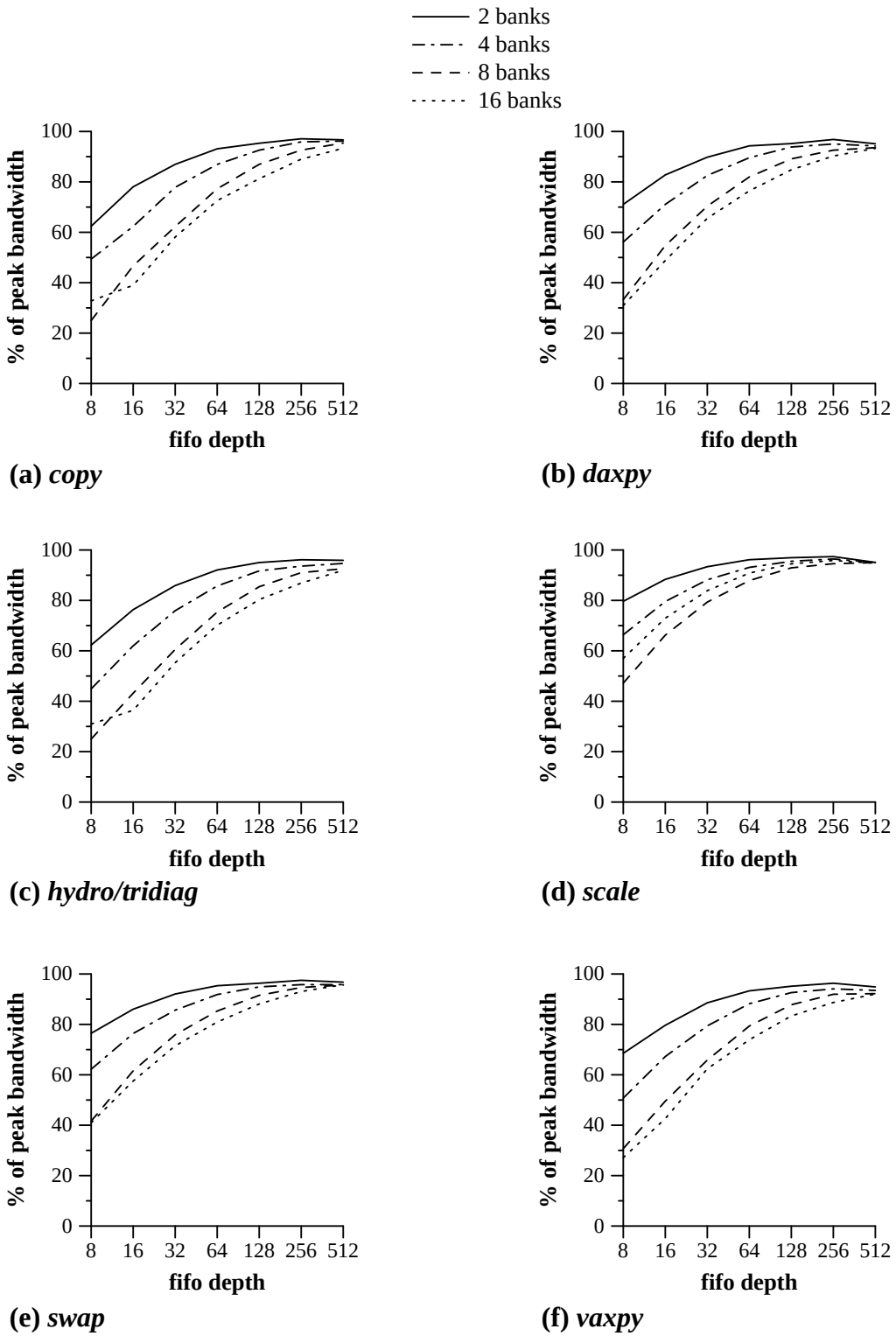



**Figure 31 Prescheduled Exhaustive BC Performance for 2 CEs**

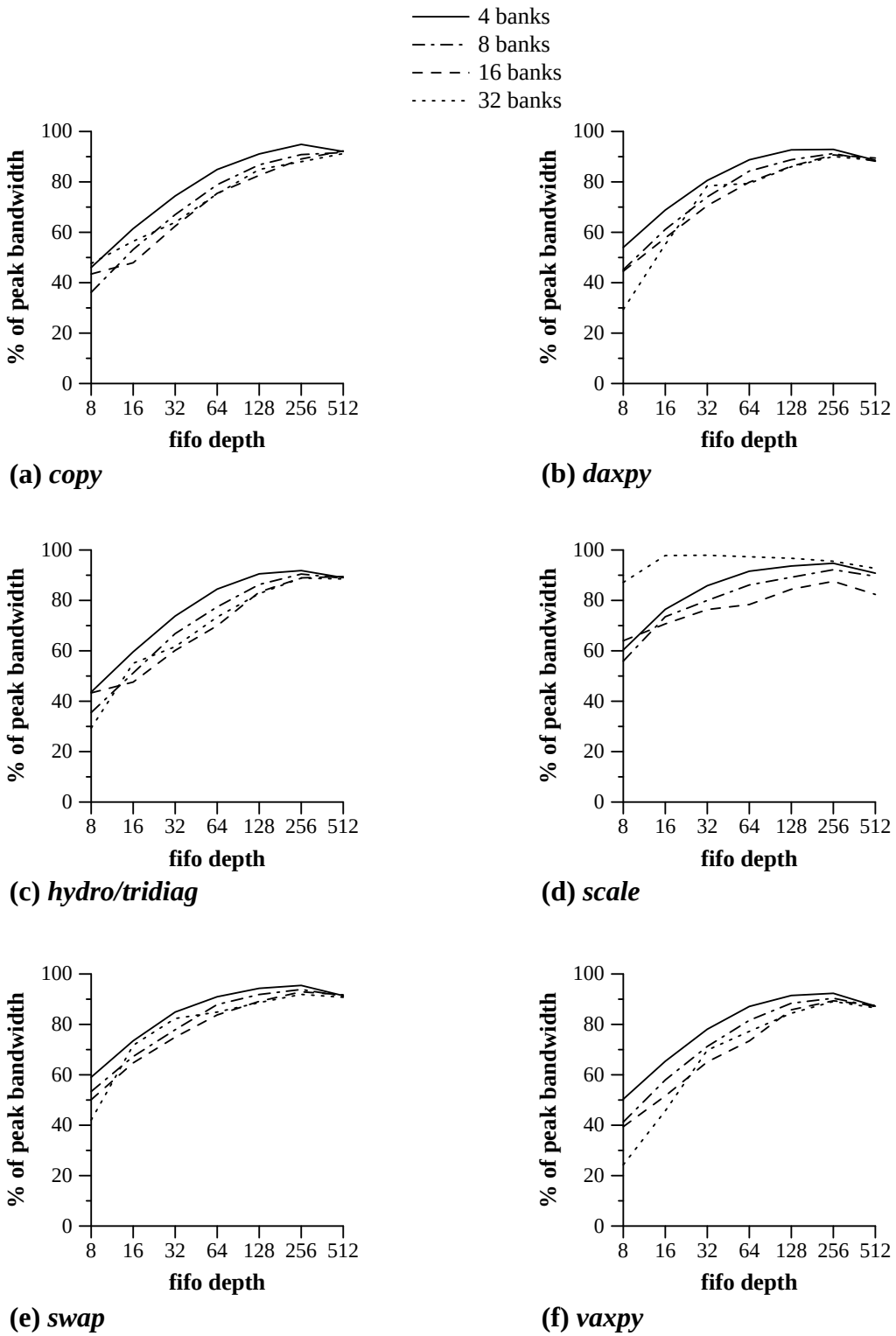



**Figure 32 Prescheduled Exhaustive BC Performance for 4 CEs**

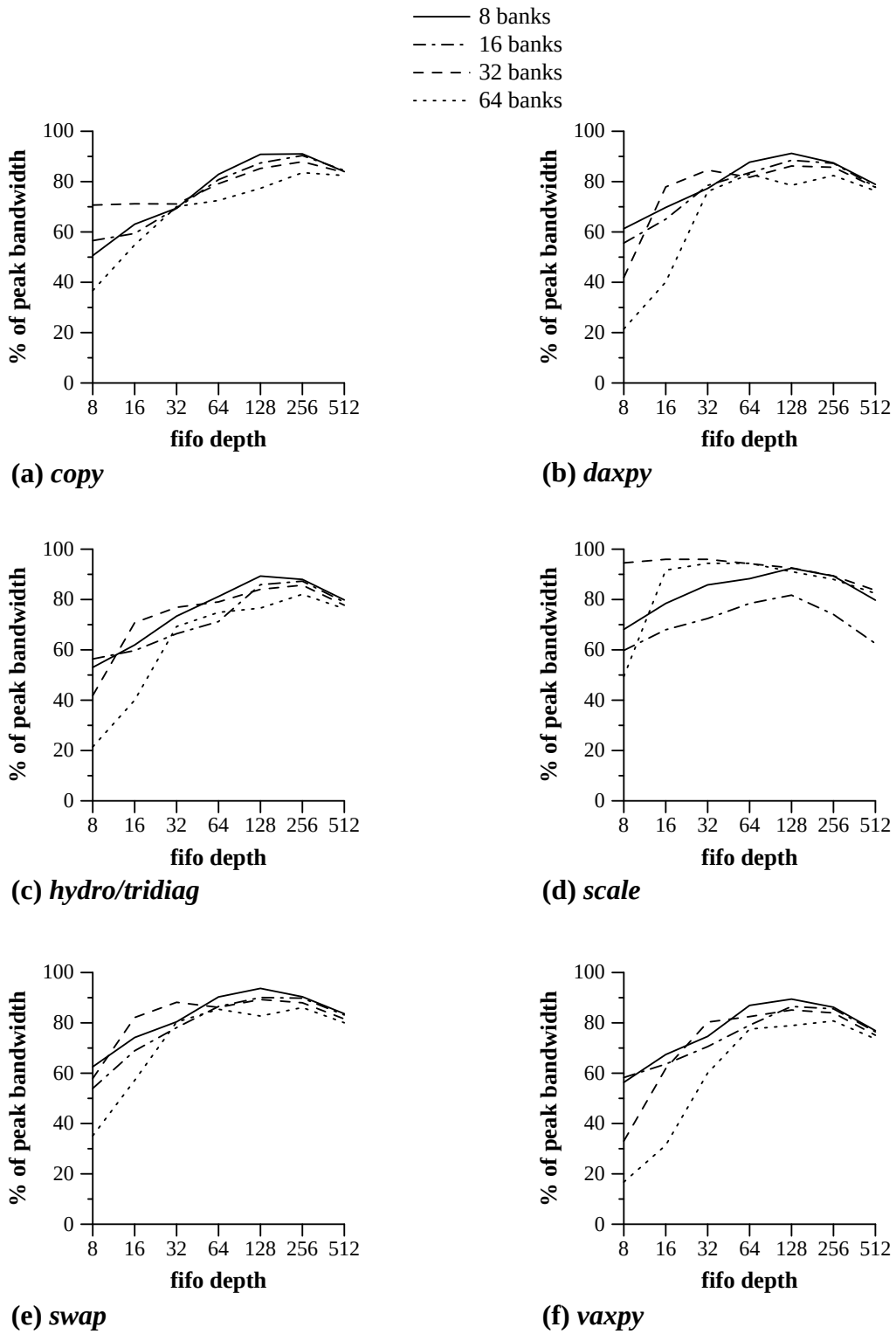
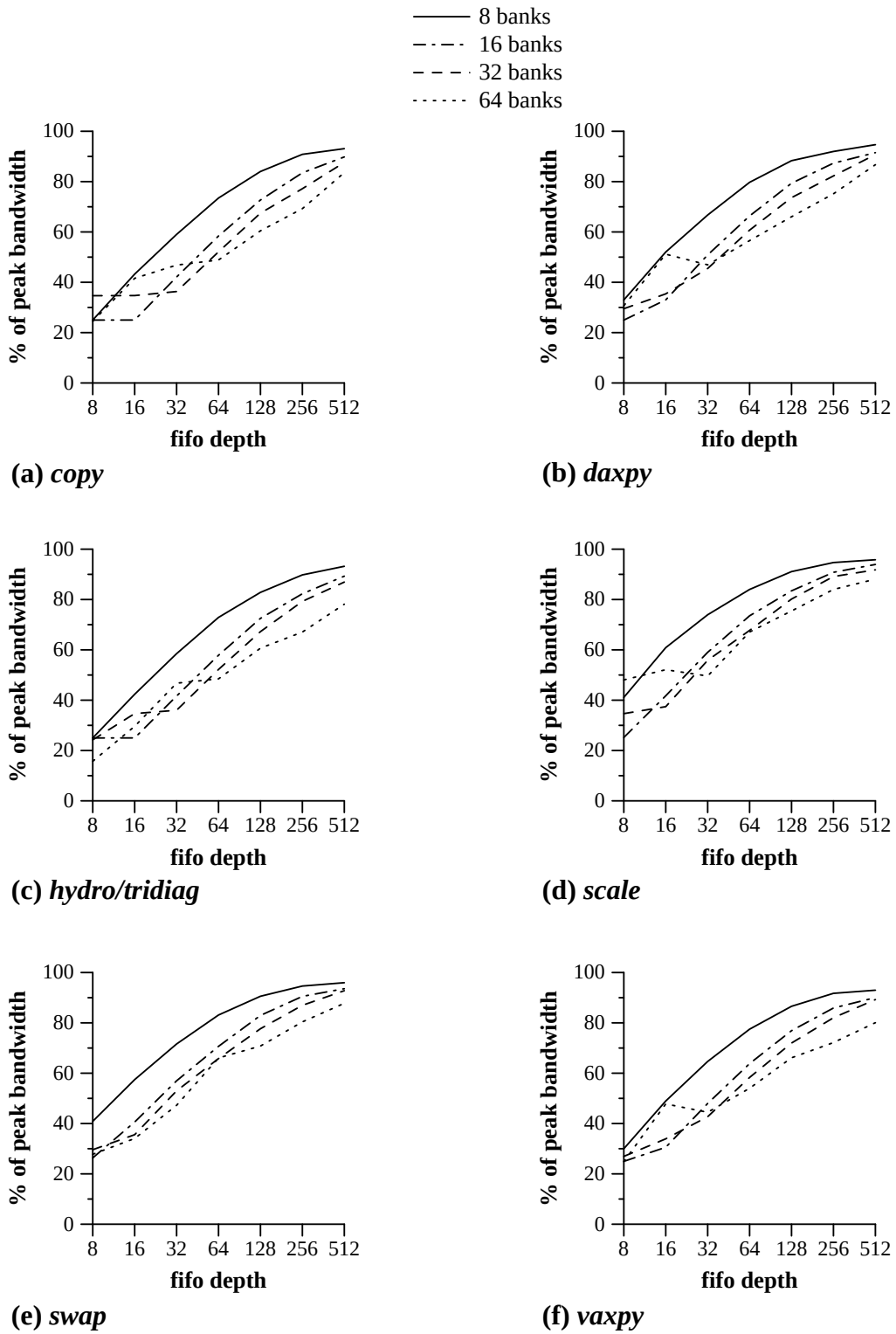



**Figure 33 Prescheduled Exhaustive BC Performance for 8 CEs**

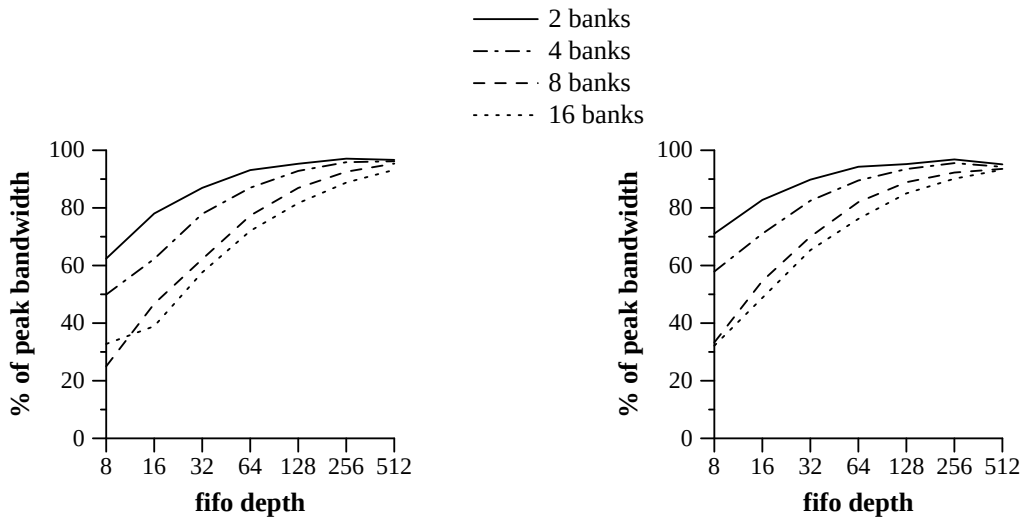



**(a) *copy*** **(b) *daxpy***

**(c) *hydro/tridiag*** **(d) *scale***

**(e) *swap*** **(f) *vaxpy***

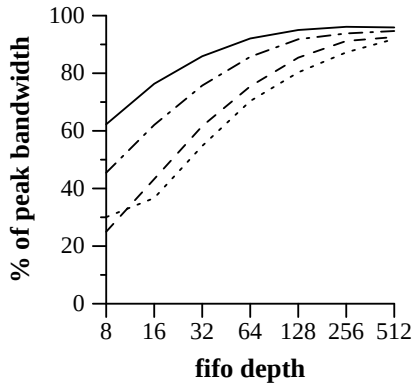
**Figure 34 Prescheduled Exhaustive BC Performance for 8 CEs (Longer Vectors)**



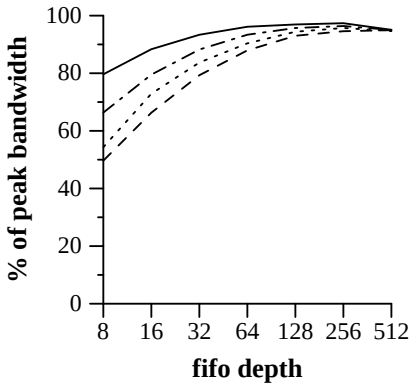

**(a)** ***copy***



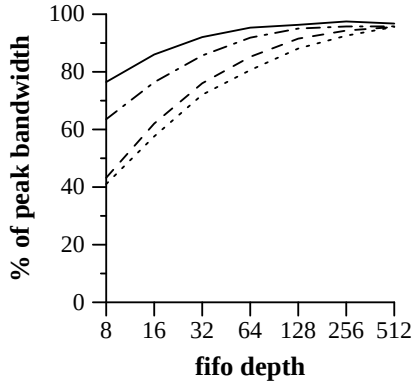

**(b)** ***daxpy***



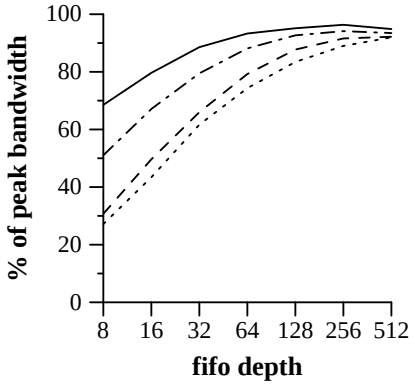

**(c)** ***hydro/tridiag***




**(d)** ***scale***




**(e)** ***swap***




**(f)** ***vaxpy***

**Figure 35 Prescheduled Token BC Performance for 2 CEs (Modular Bank Sets)**

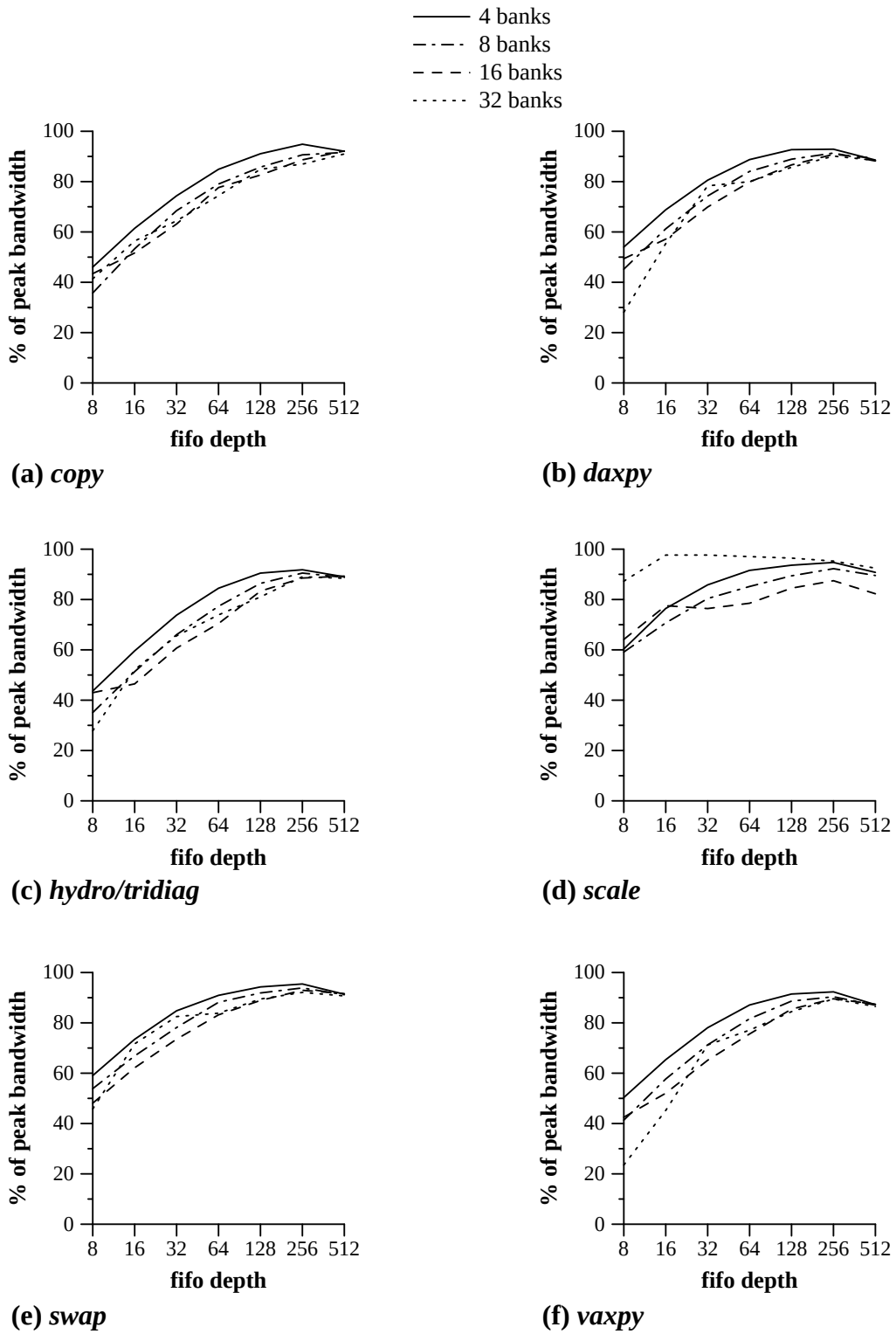
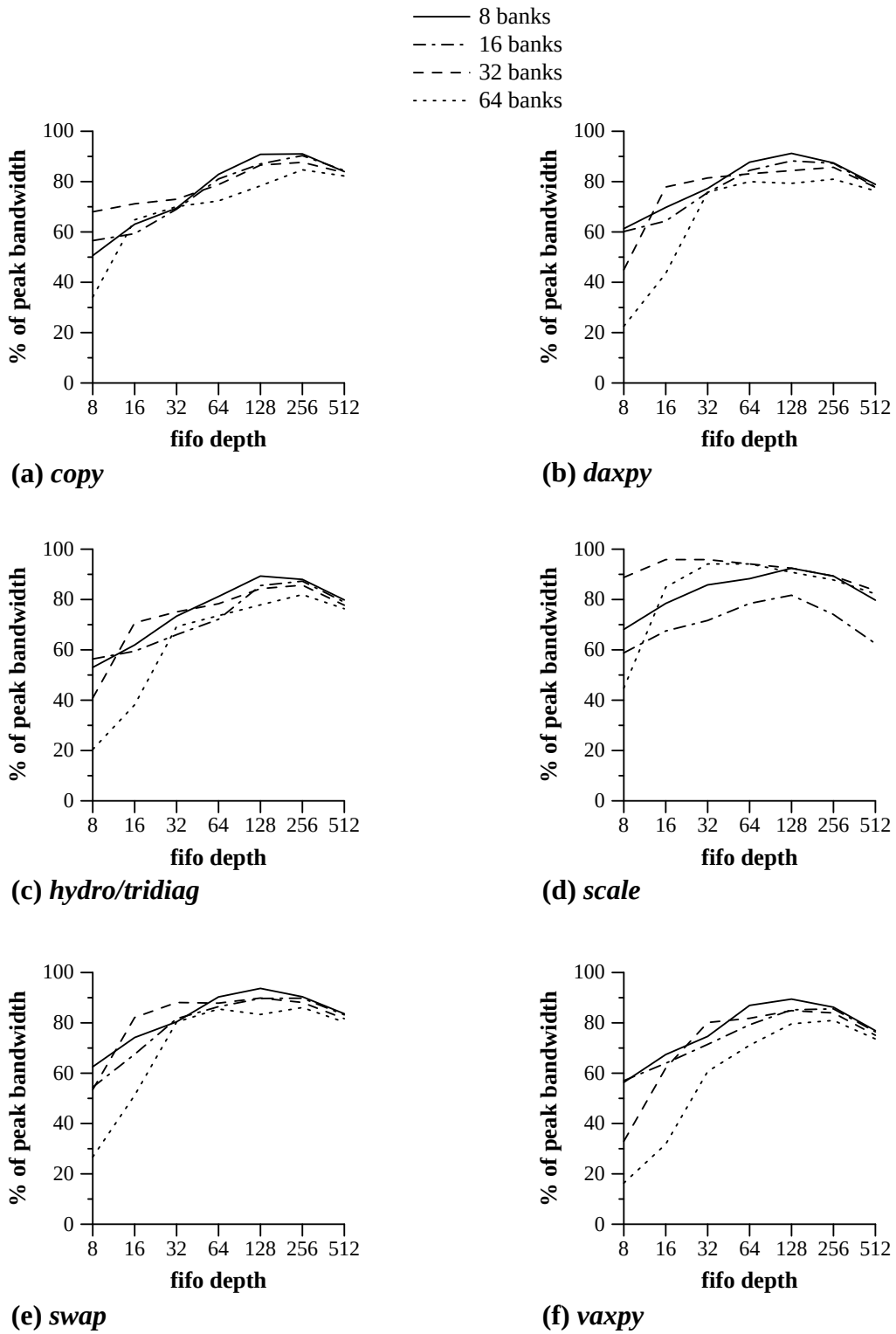



**Figure 36 Prescheduled Token BC Performance for 4 CEs (Modular Bank Sets)**




**(a) *copy*** **(b) *daxpy***

**(c) *hydro/tridiag*** **(d) *scale***

**(e) *swap*** **(f) *vaxpy***

**Figure 37 Prescheduled Token BC Performance for 8 CEs (Modular Bank Sets)**

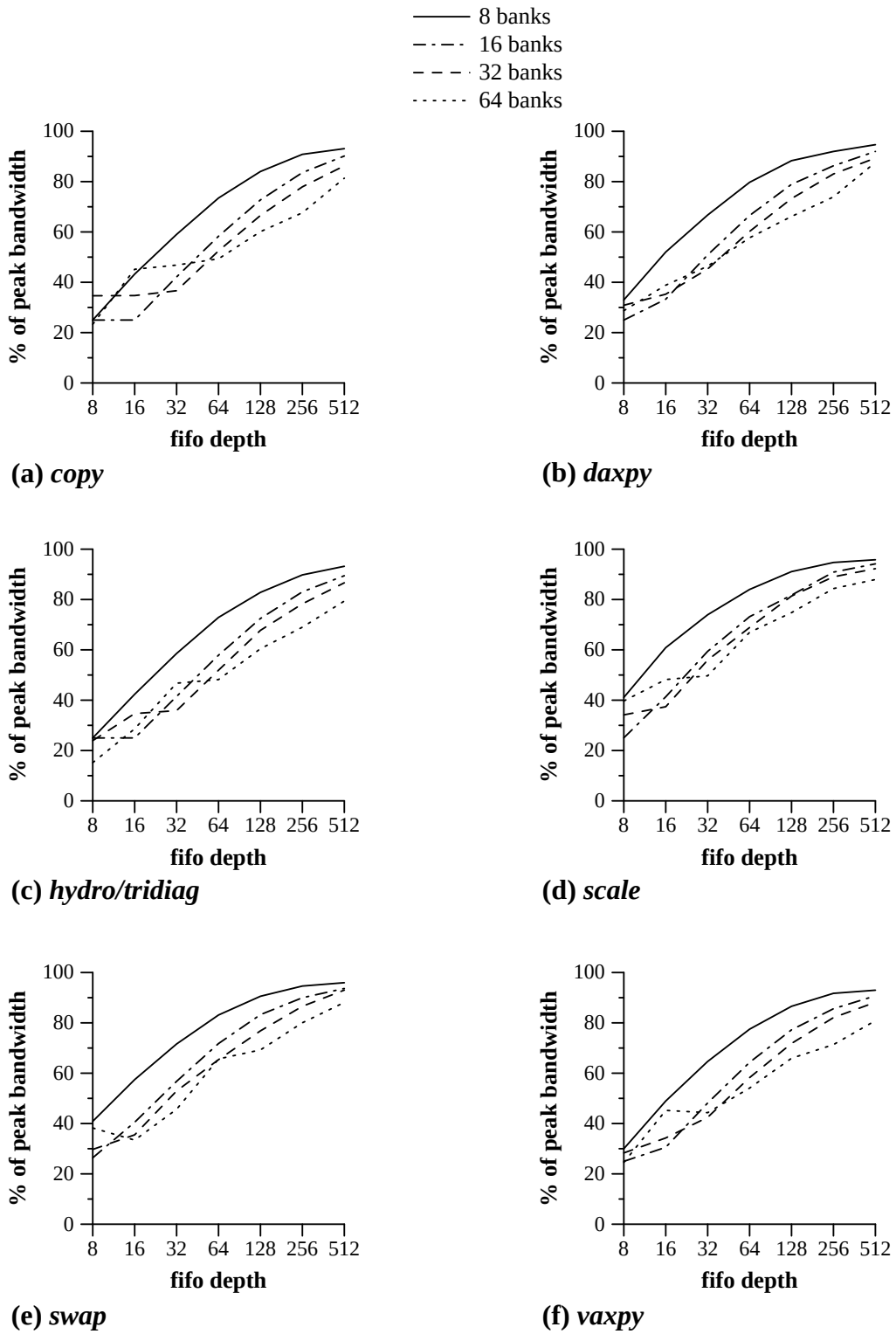



**Figure 38 Prescheduled Token BC Performance for 8 CEs (Longer Vectors, Modular Bank Sets)**

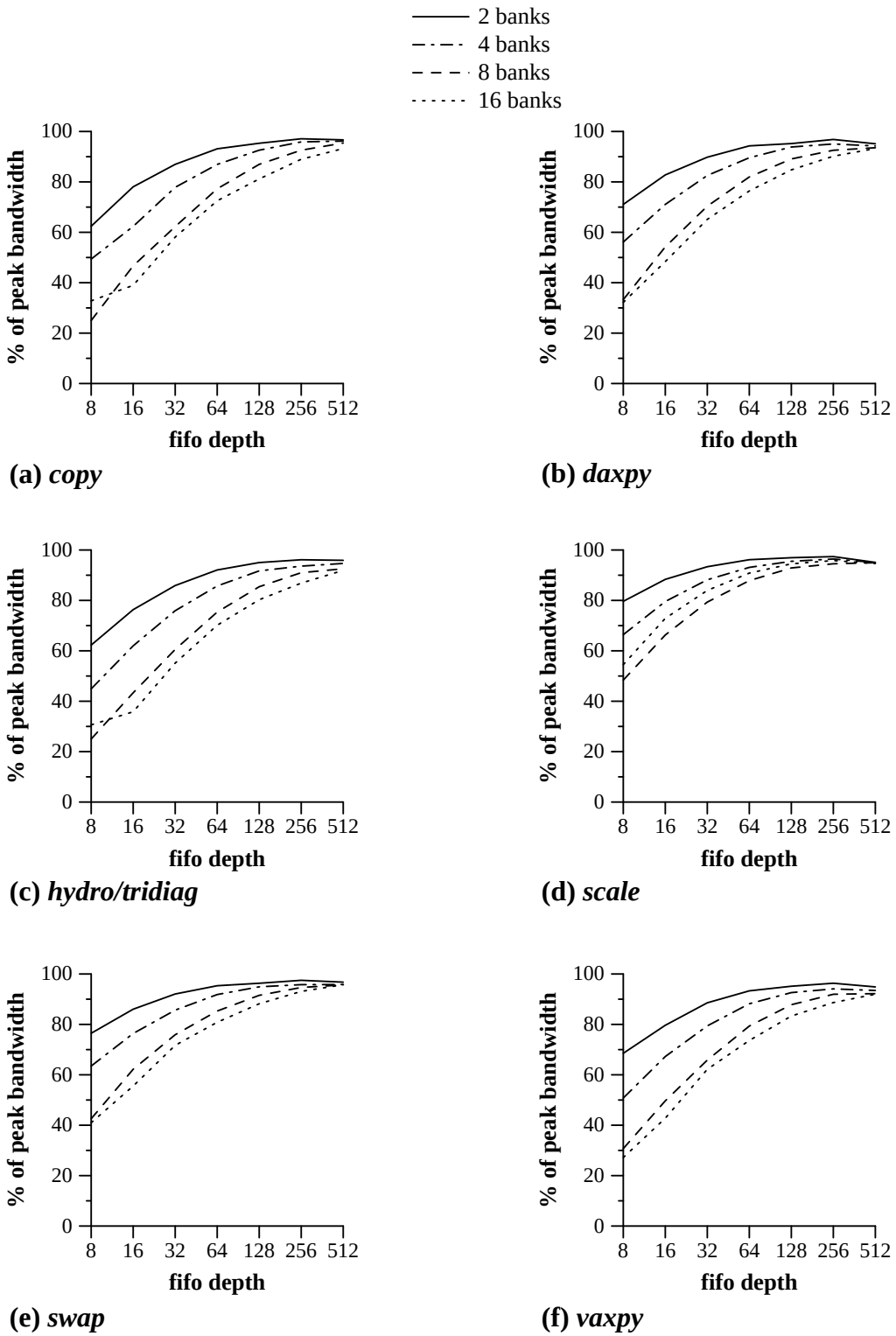



**Figure 39 Prescheduled Token BC Performance for 2 CEs**

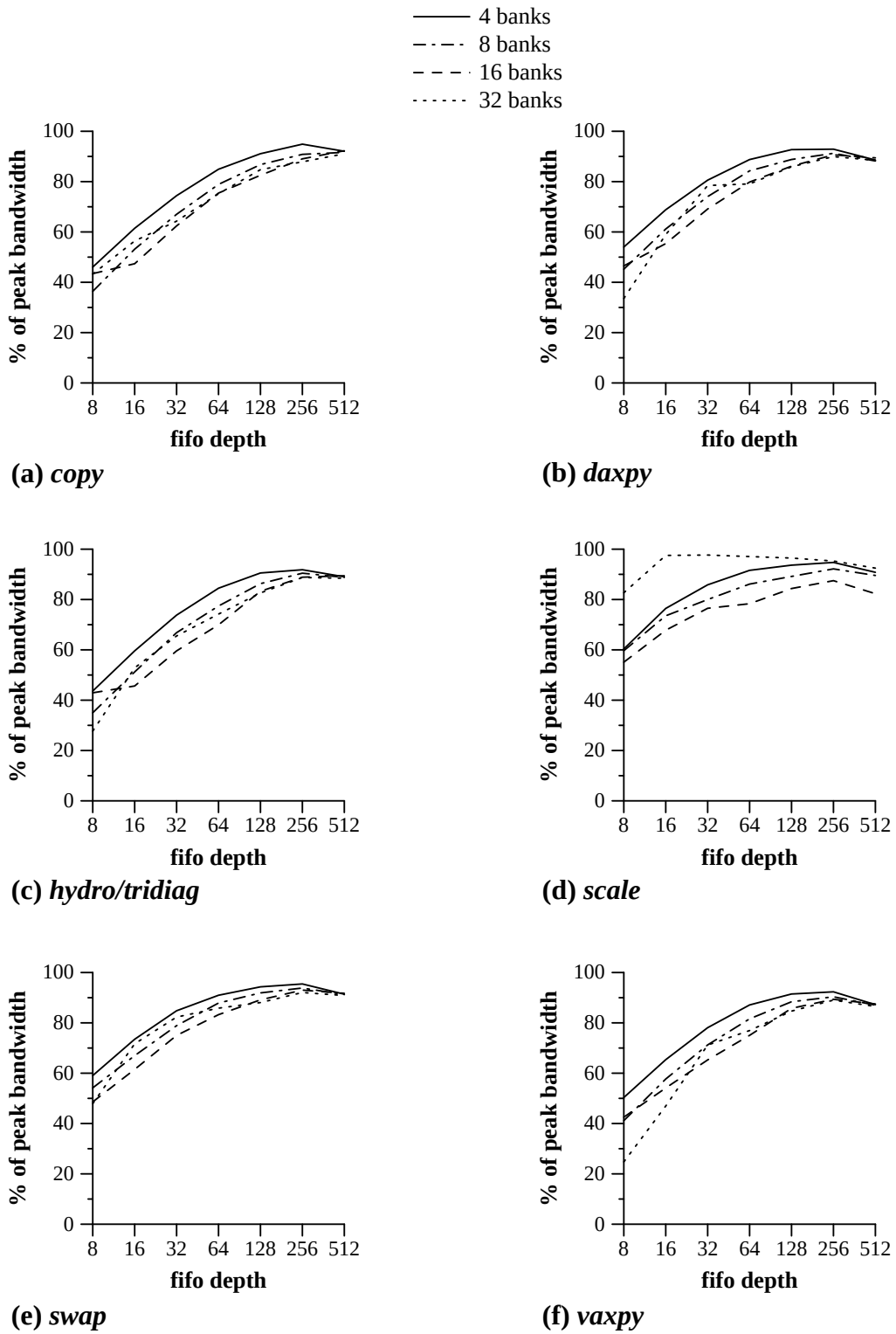



**Figure 40 Prescheduled Token BC Performance for 4 CEs**

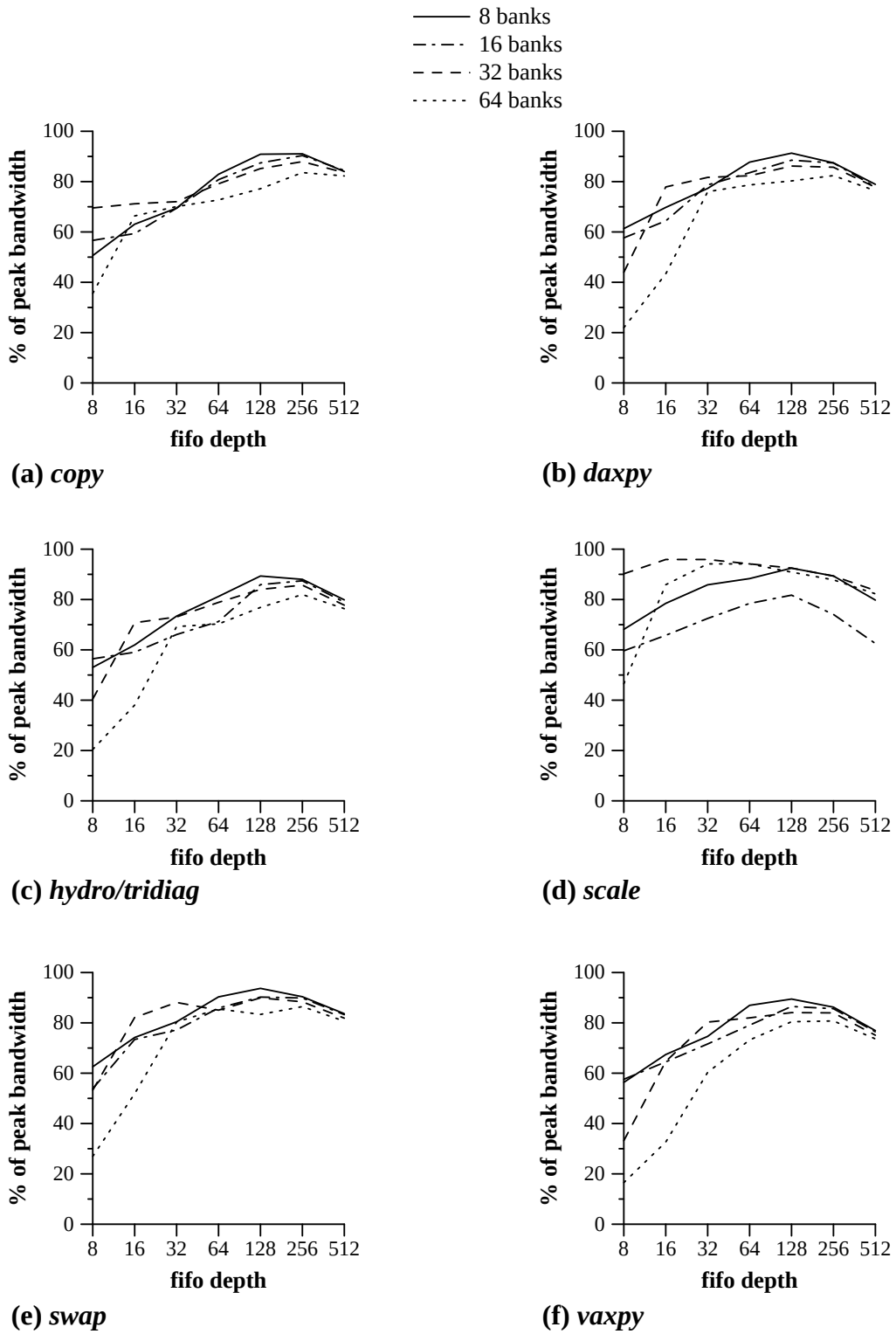



**Figure 41 Prescheduled Token BC Performance for 8 CEs**

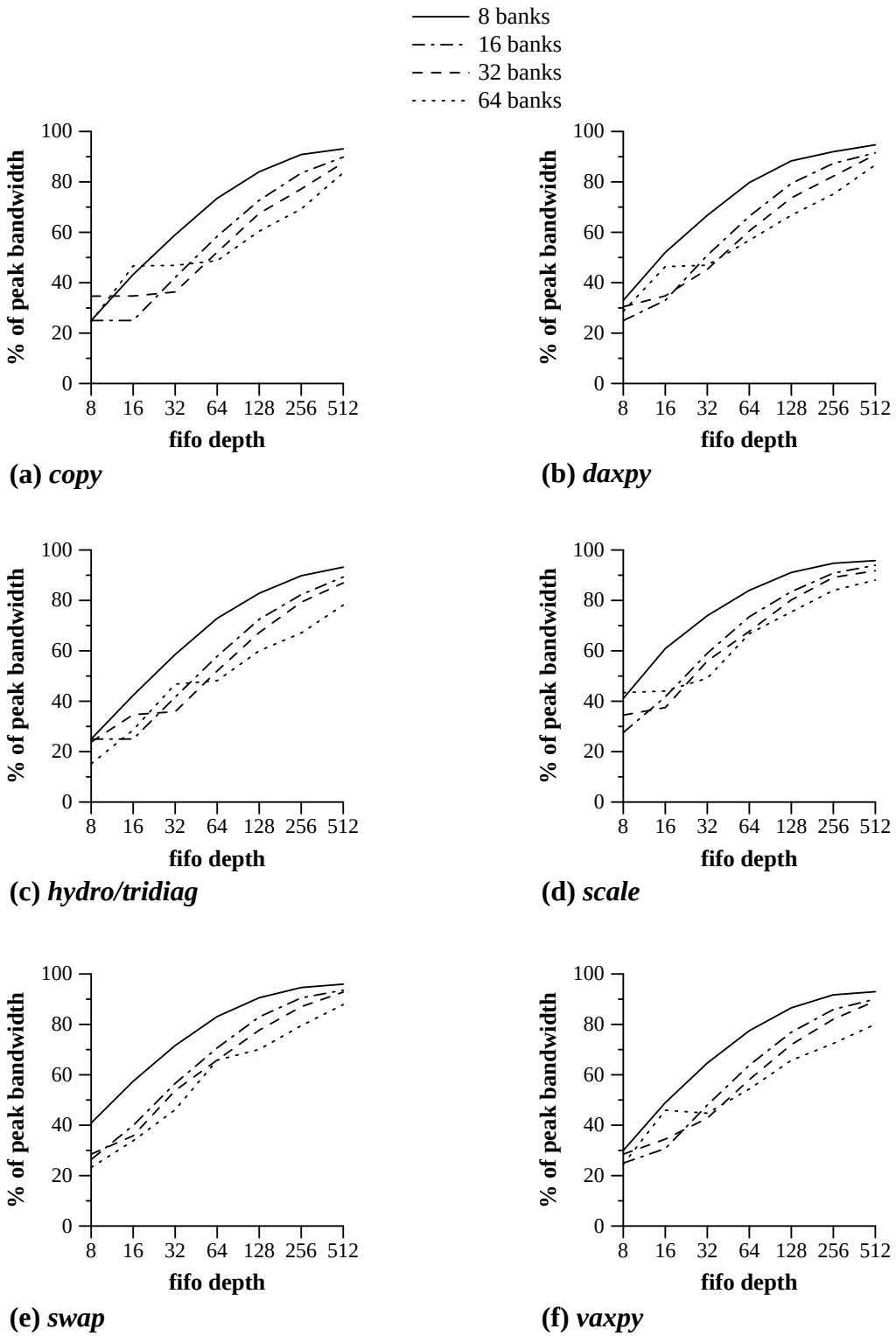



**Figure 42 Prescheduled Token BC Performance for 8 CEs (Longer Vectors)**

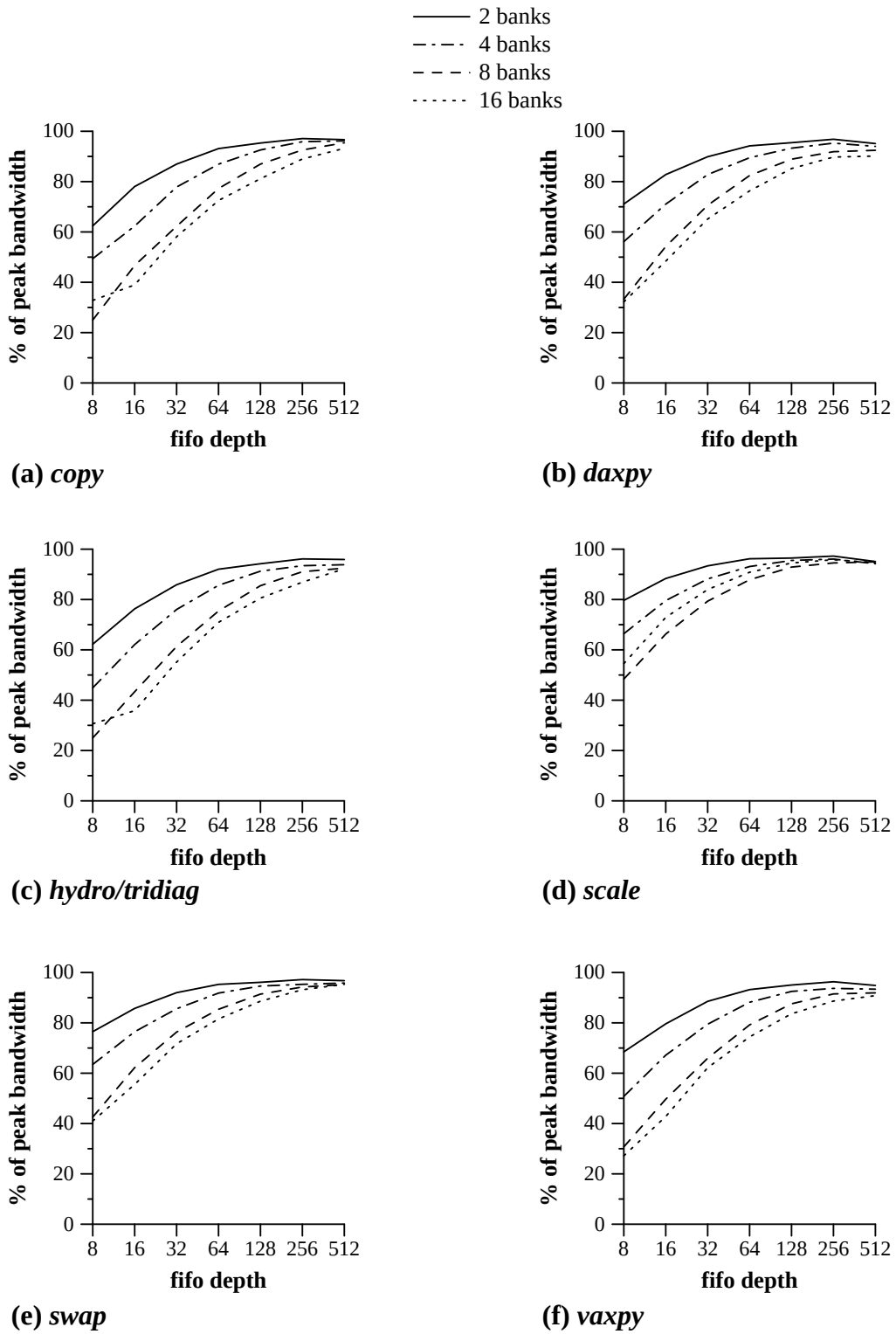



**Figure 43 Prescheduled Token TBC Performance for 2 CEs**

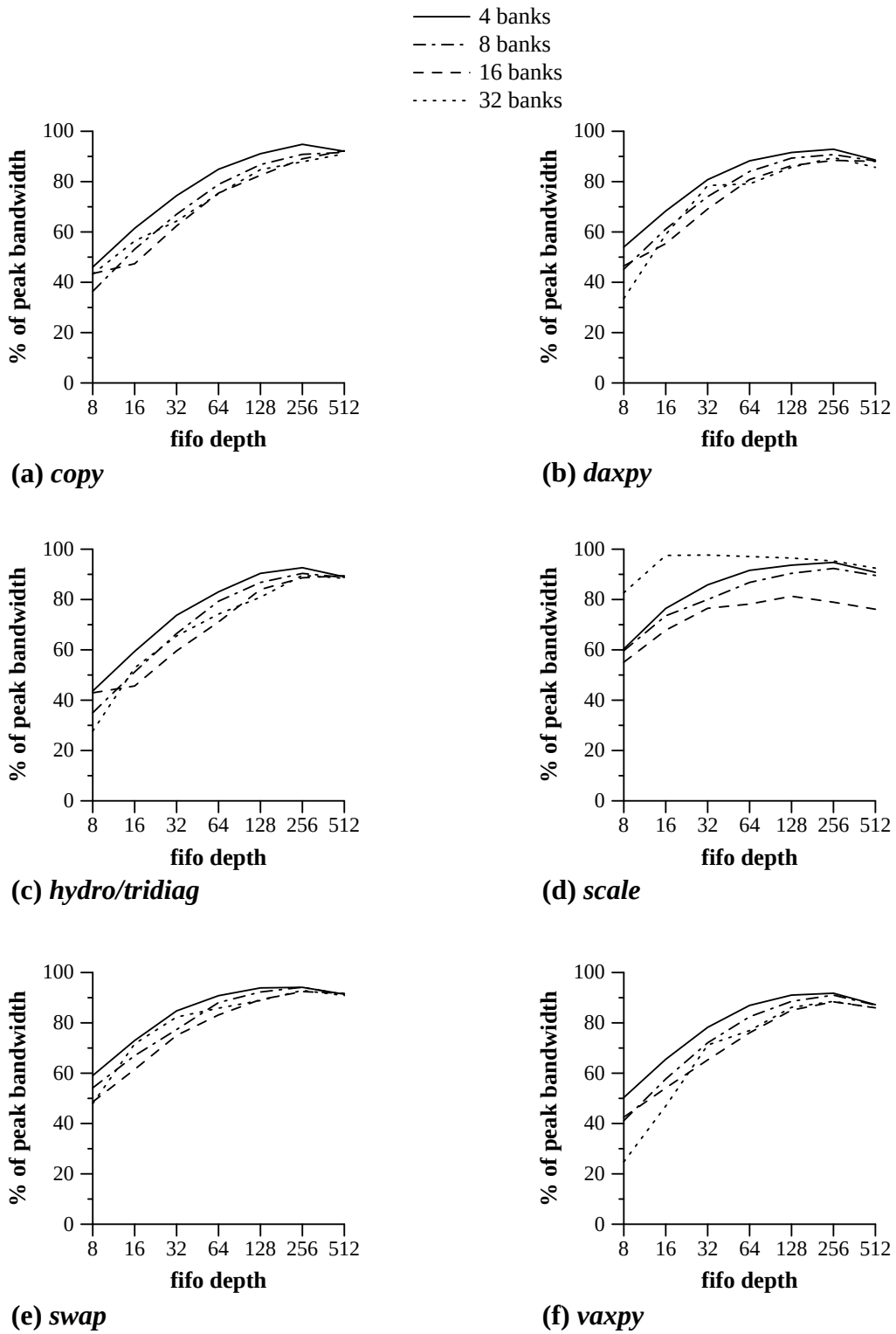



**Figure 44 Prescheduled Token TBC Performance for 4 CEs**

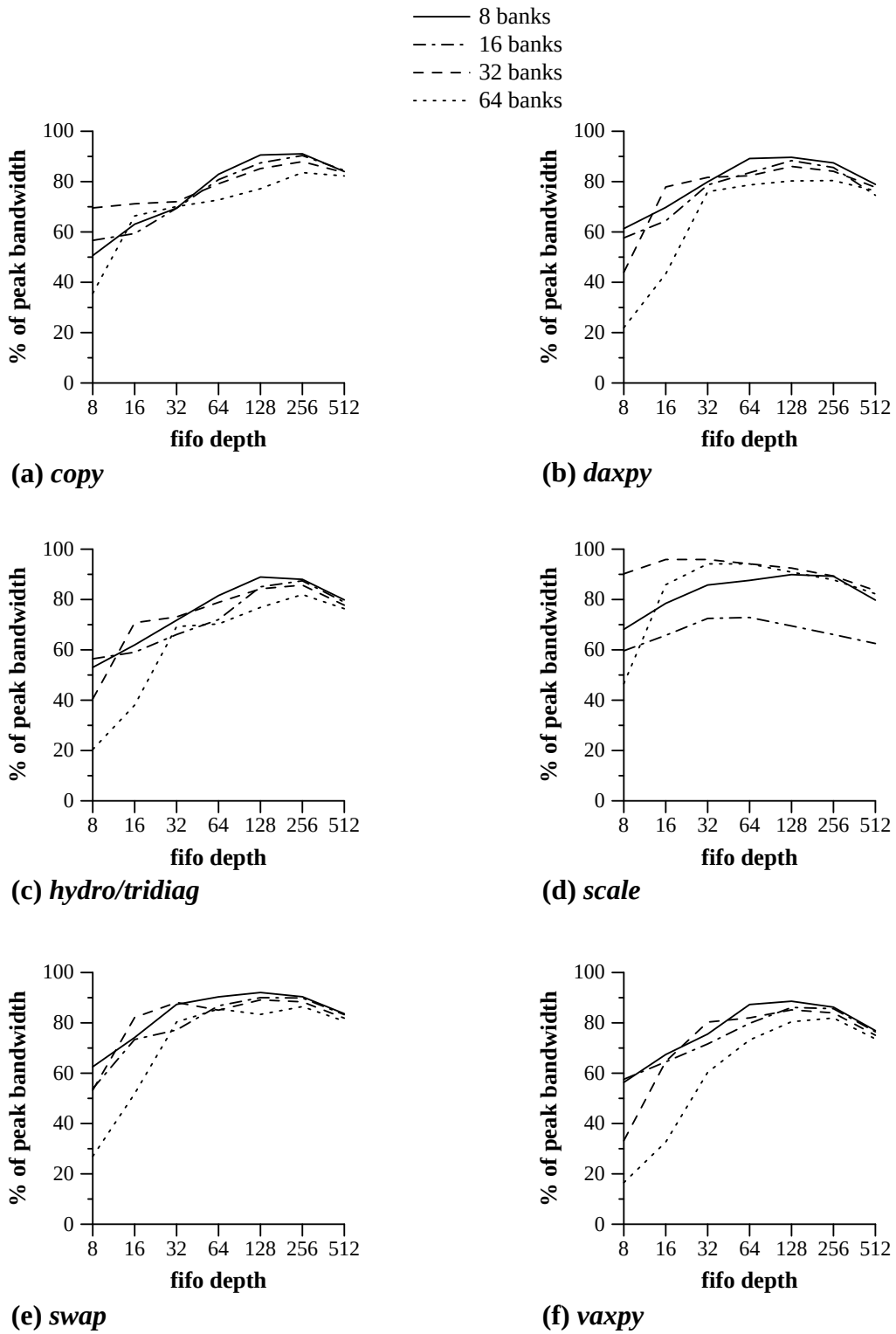



**Figure 45 Prescheduled Token TBC Performance for 8 CEs**

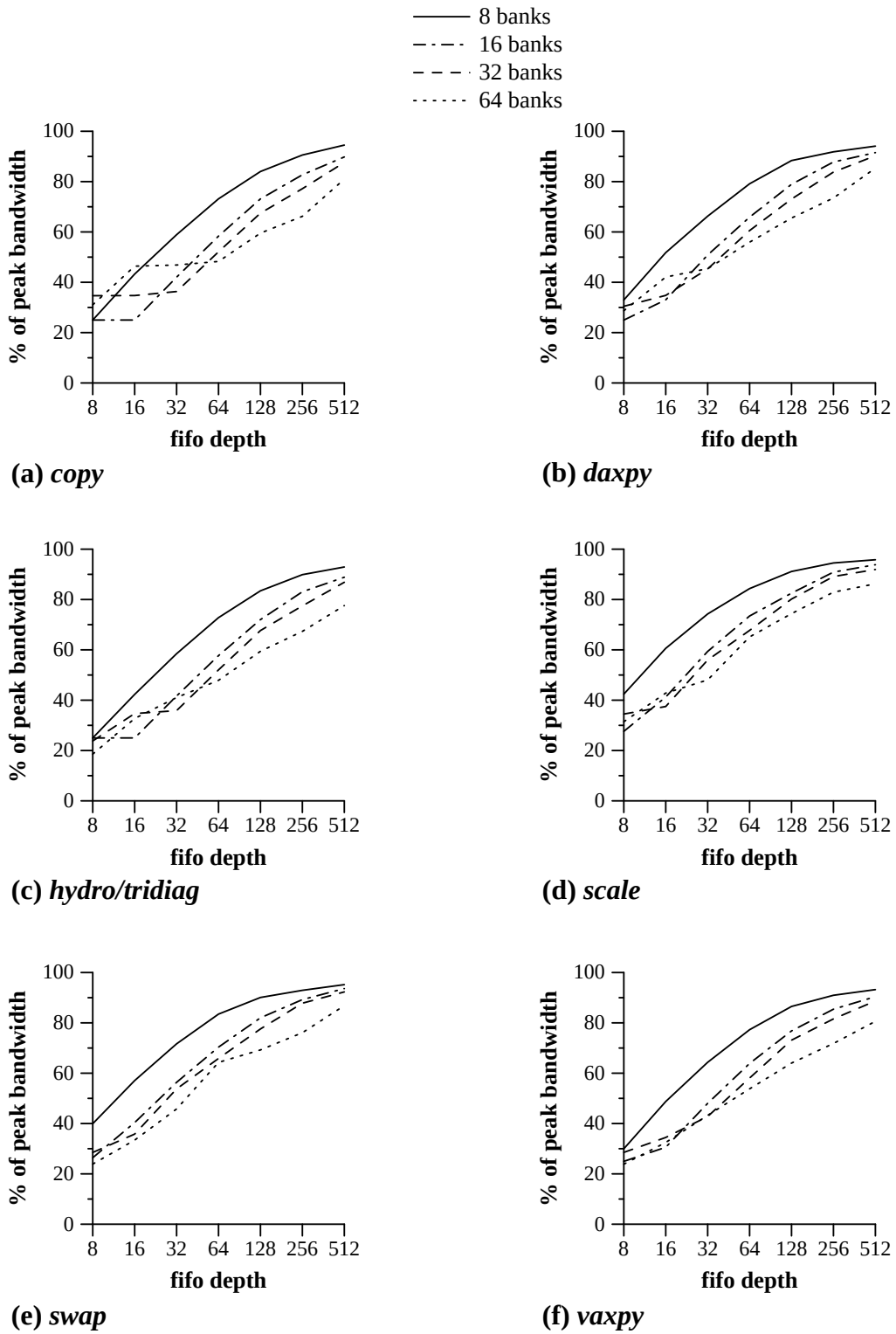



**Figure 46 Prescheduled Token TBC Performance for 8 CEs (Longer Vectors)**

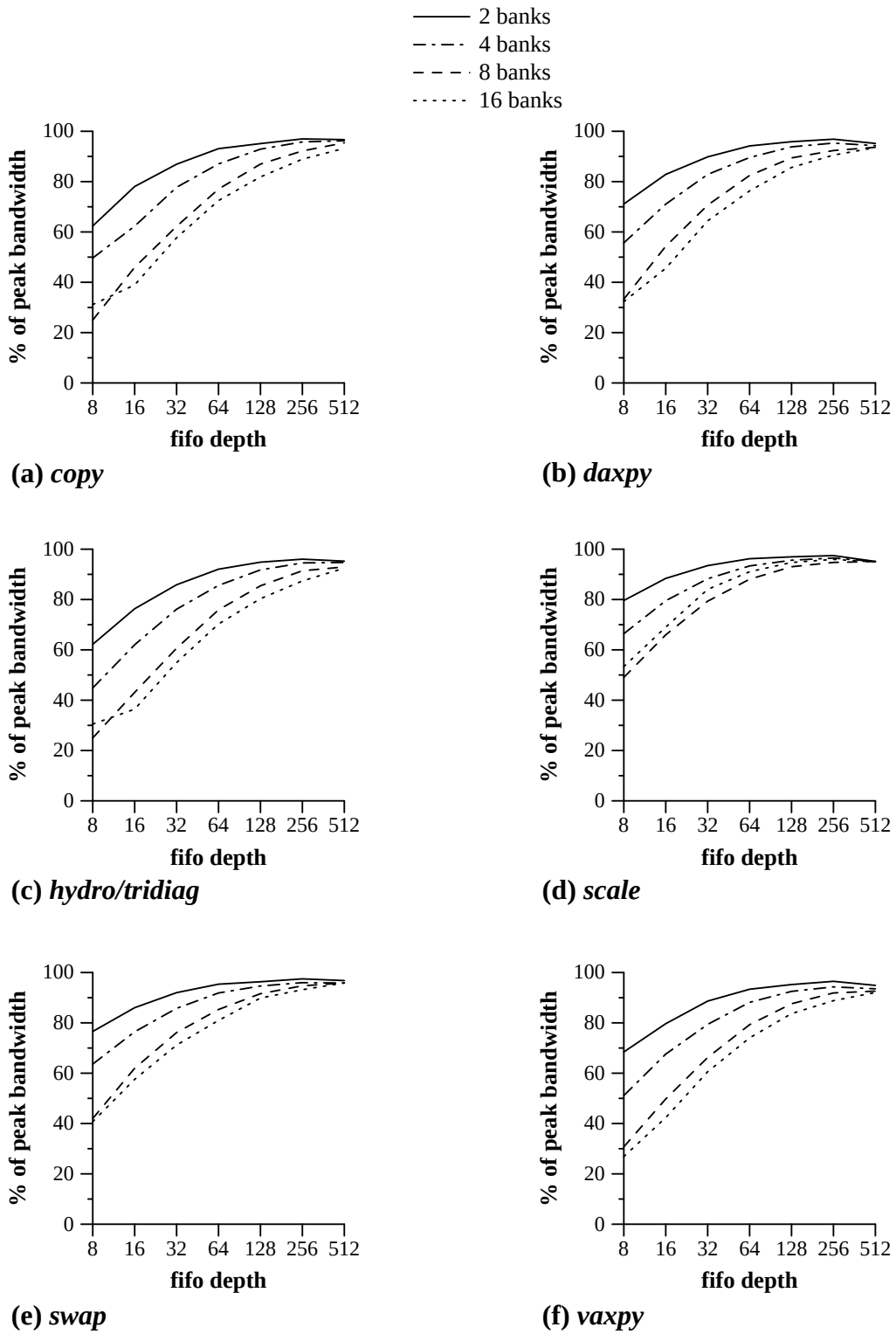



**Figure 47 Prescheduled Token BC Performance for 2 CEs (Staggered Alignment)**

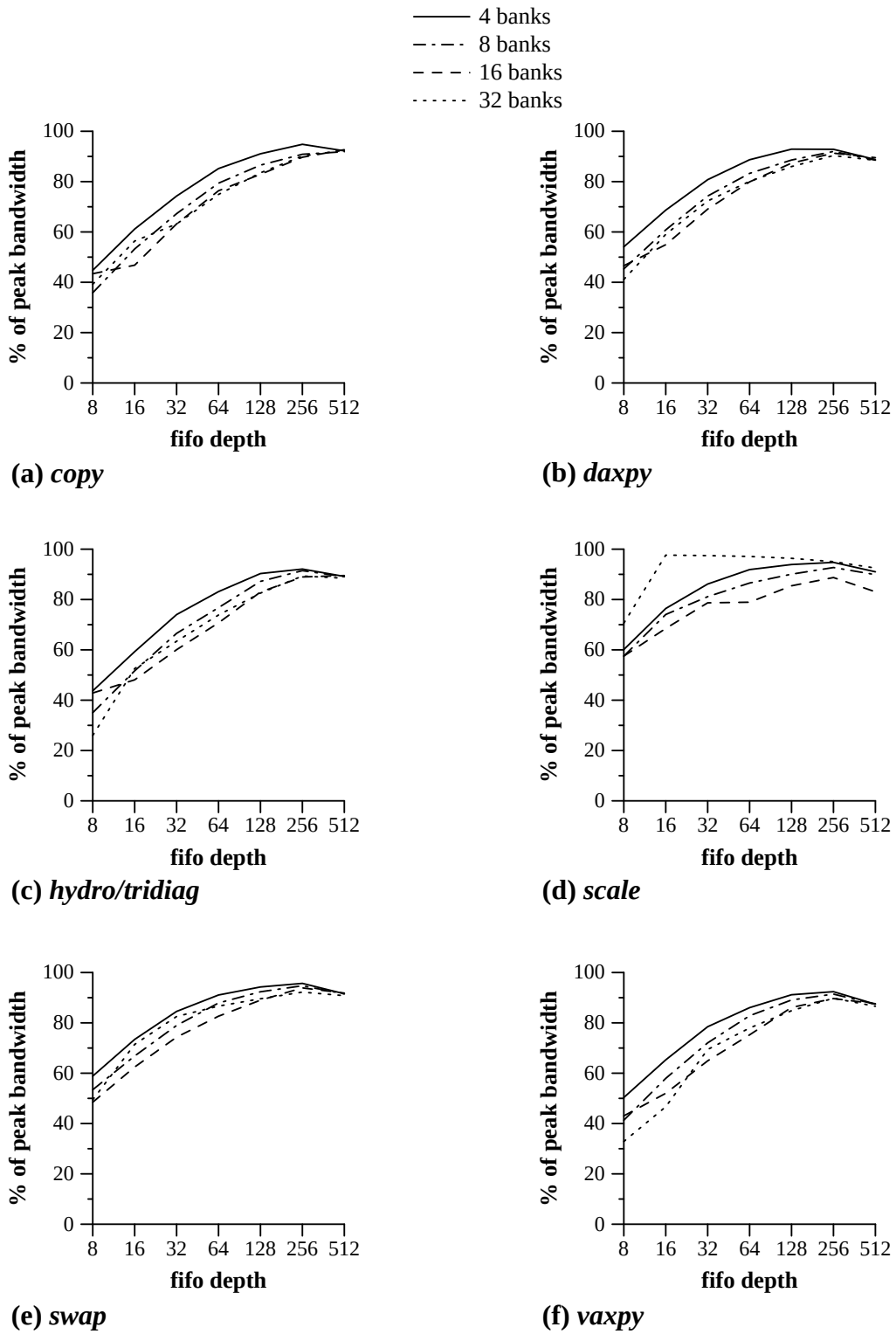
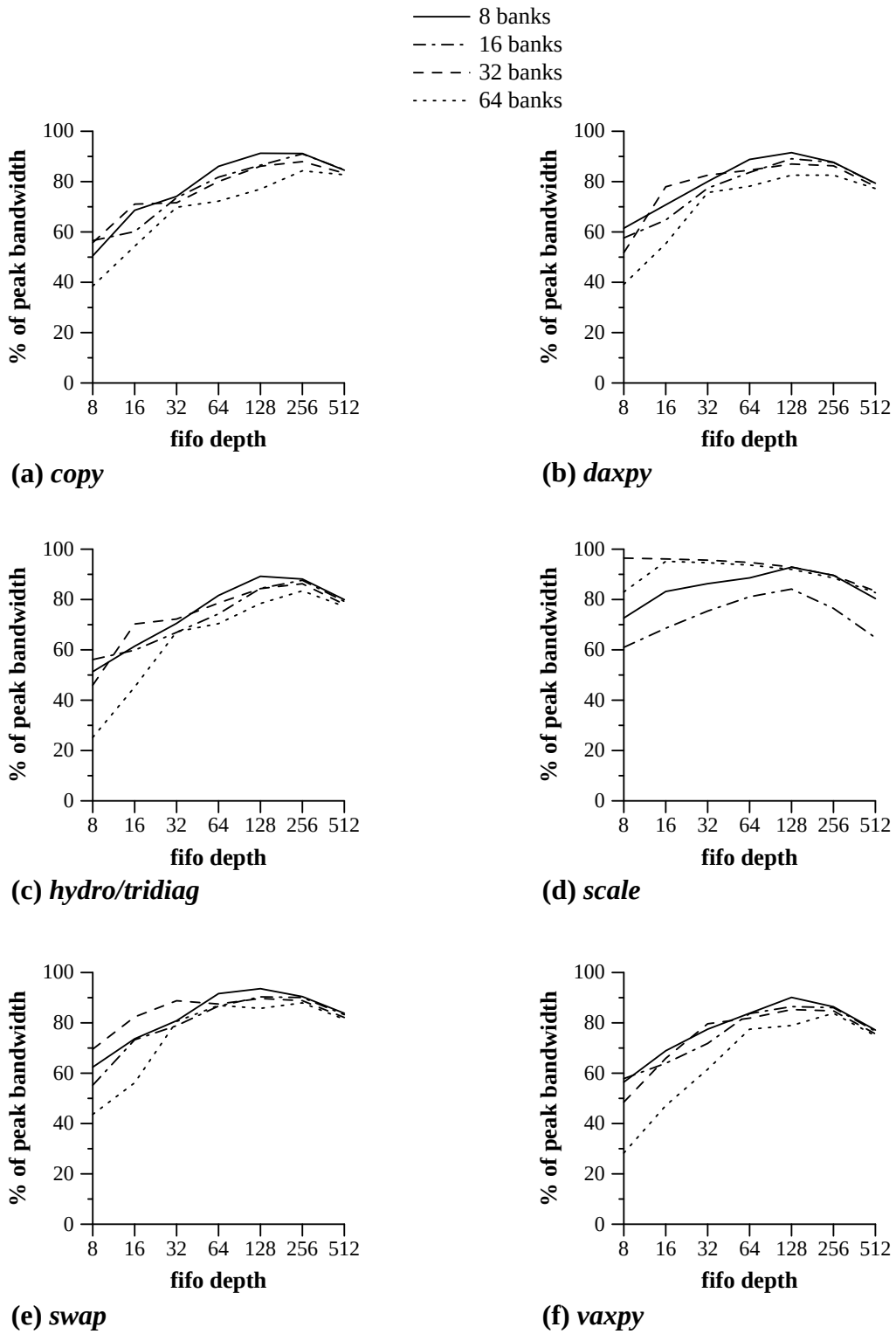



**Figure 48 Prescheduled Token BC Performance for 4 CEs (Staggered Alignment)**




**(a) *copy*** **(b) *daxpy***

**(c) *hydro/tridiag*** **(d) *scale***

**(e) *swap*** **(f) *vaxpy***

**Figure 49 Prescheduled Token BC Performance for 8 CEs (Staggered Alignment)**

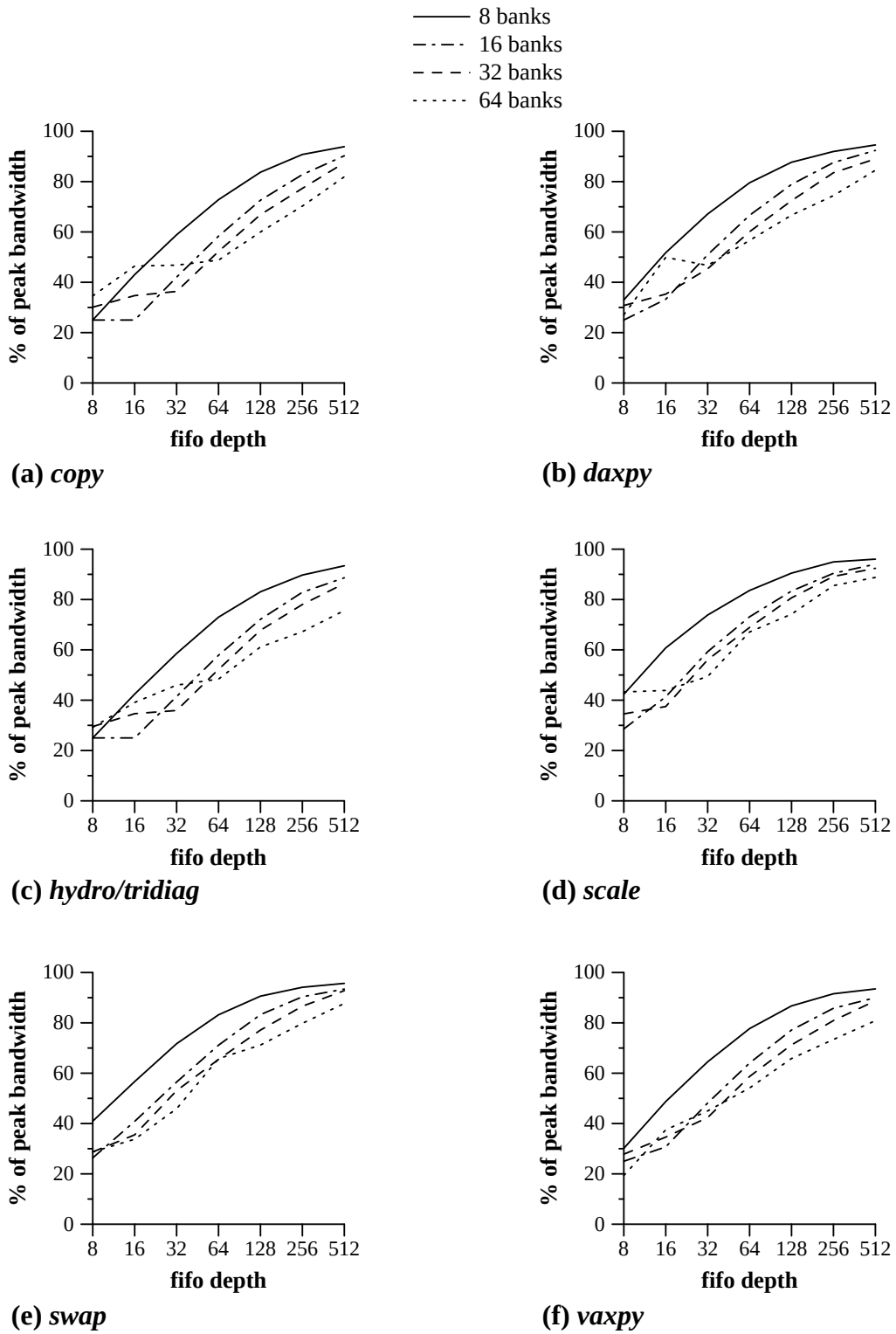



**Figure 50 Prescheduled Token BC Performance for 8 CEs (Longer Vectors, Staggered)**

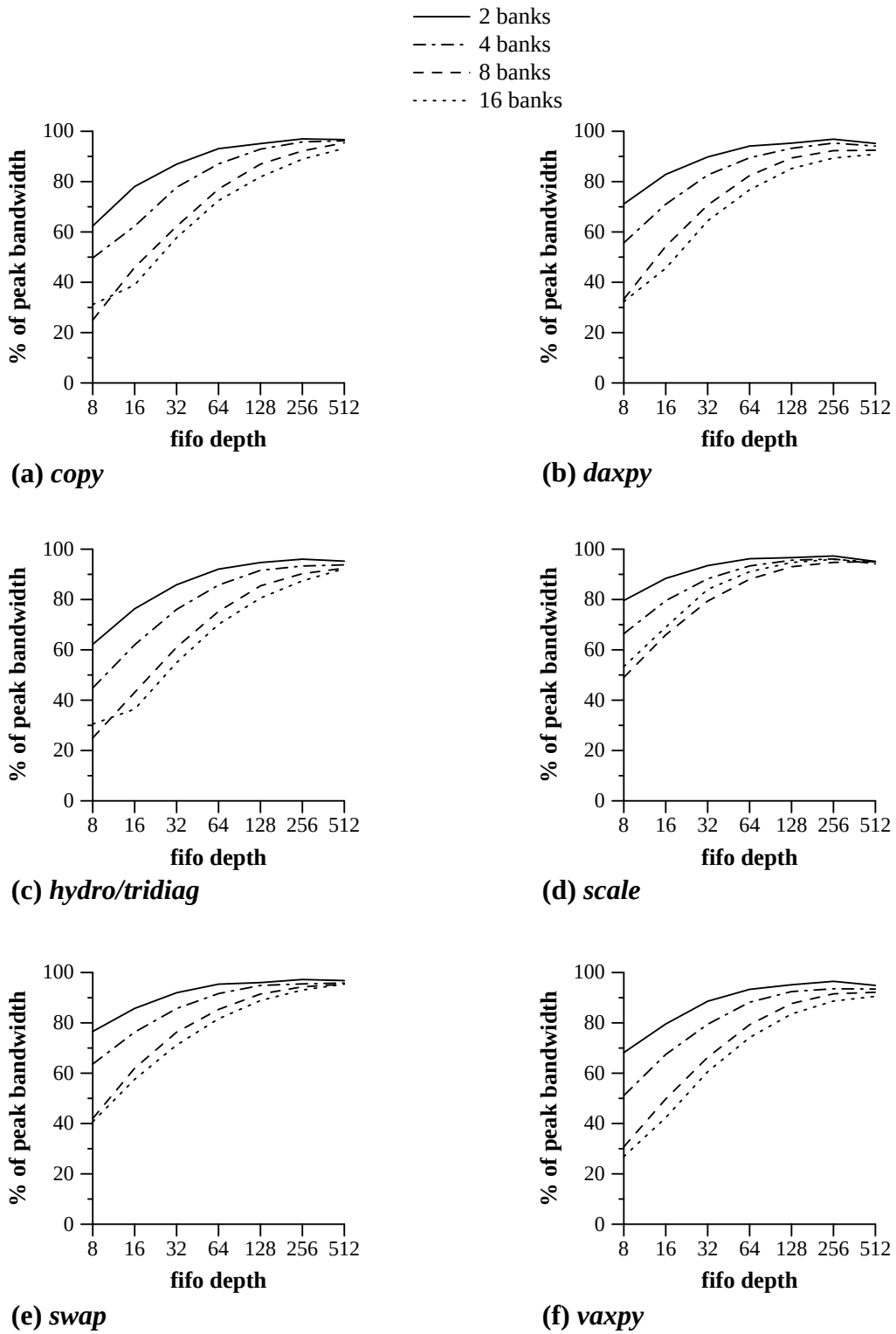



**Figure 51 Prescheduled Token TBC Performance for 2 CEs (Staggered Alignment)**

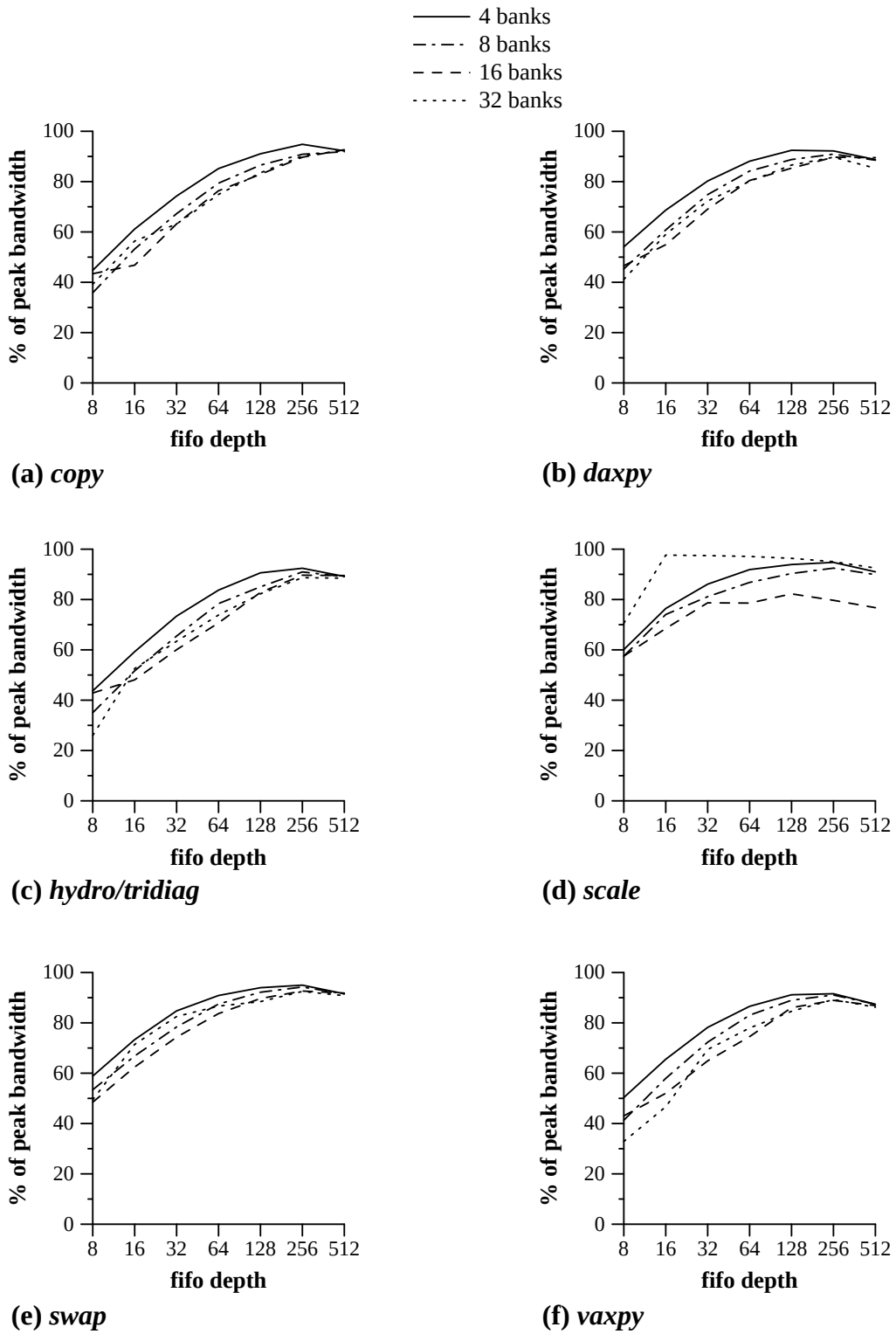



**Figure 52 Prescheduled Token TBC Performance for 4 CEs (Staggered Alignment)**

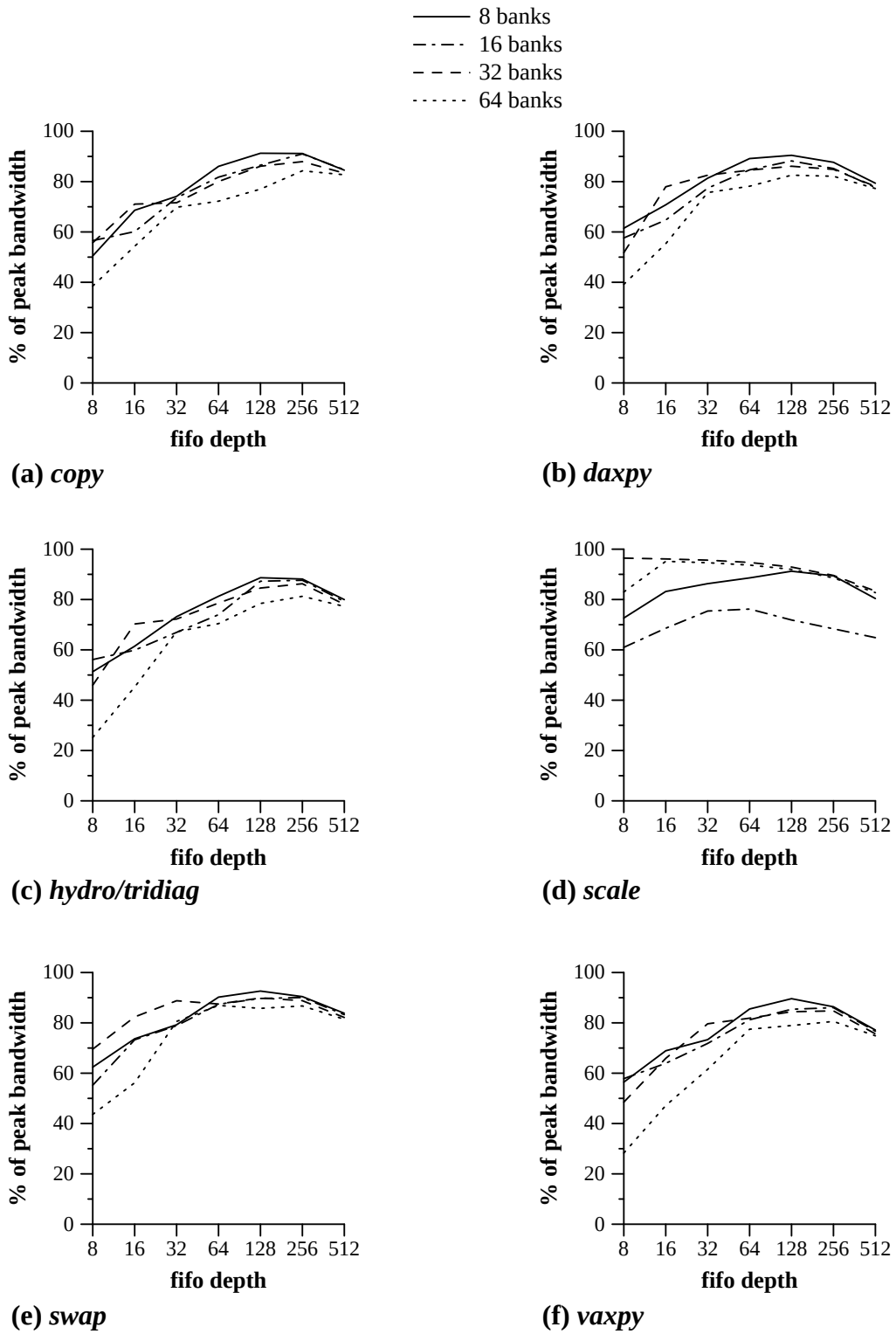



**Figure 53 Prescheduled Token TBC Performance for 8 CEs (Staggered Alignment)**

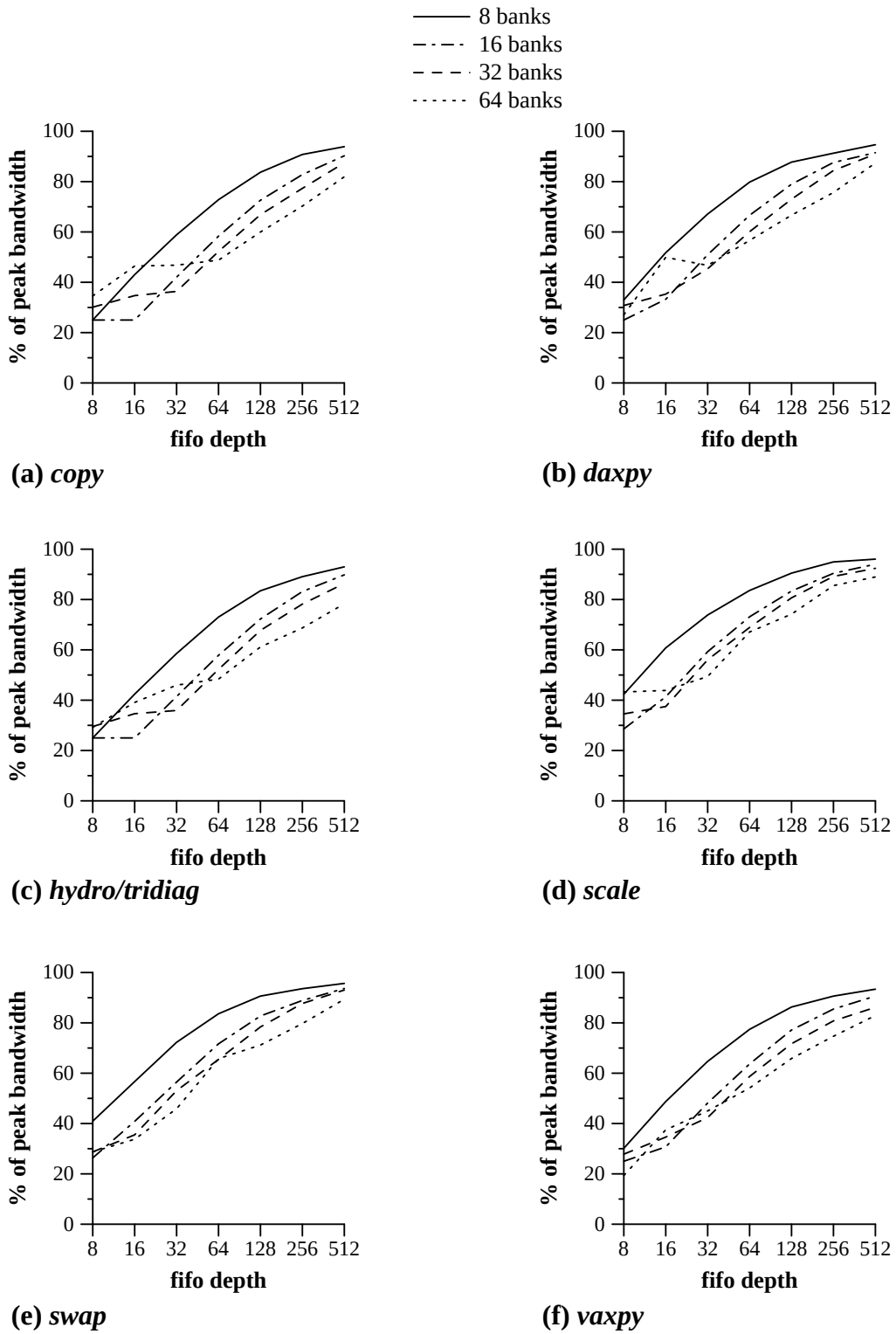
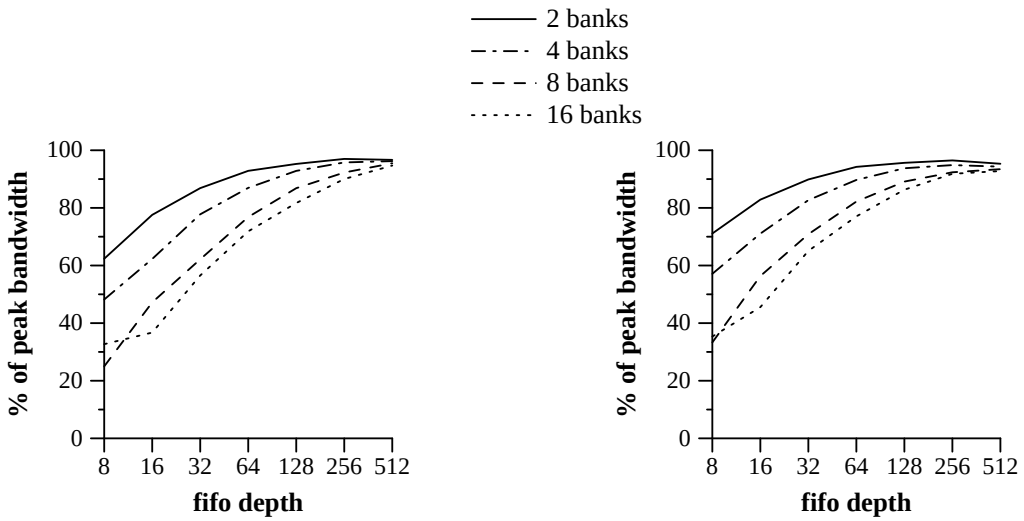


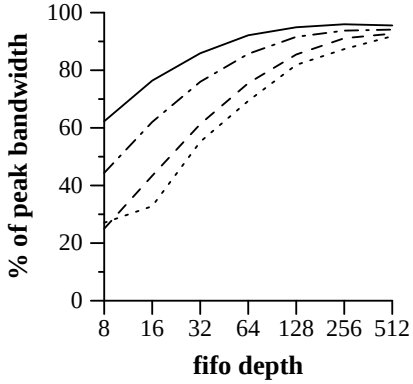

**Figure 54 Prescheduled Token TBC Performance for 8 CEs (Longer Vectors, Staggered Alignment)**



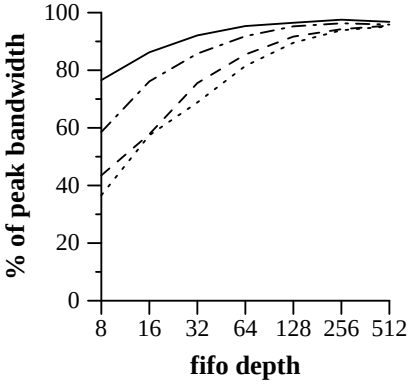

**(a)** ***copy***



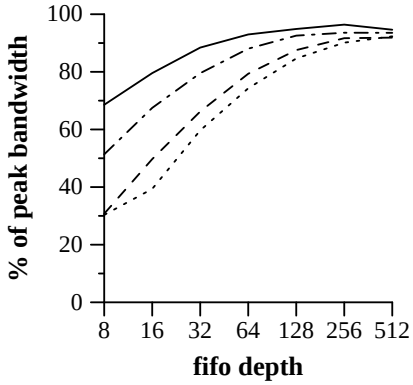

**(b)** ***daxpy***




**(c)** ***hydro/tridiag***

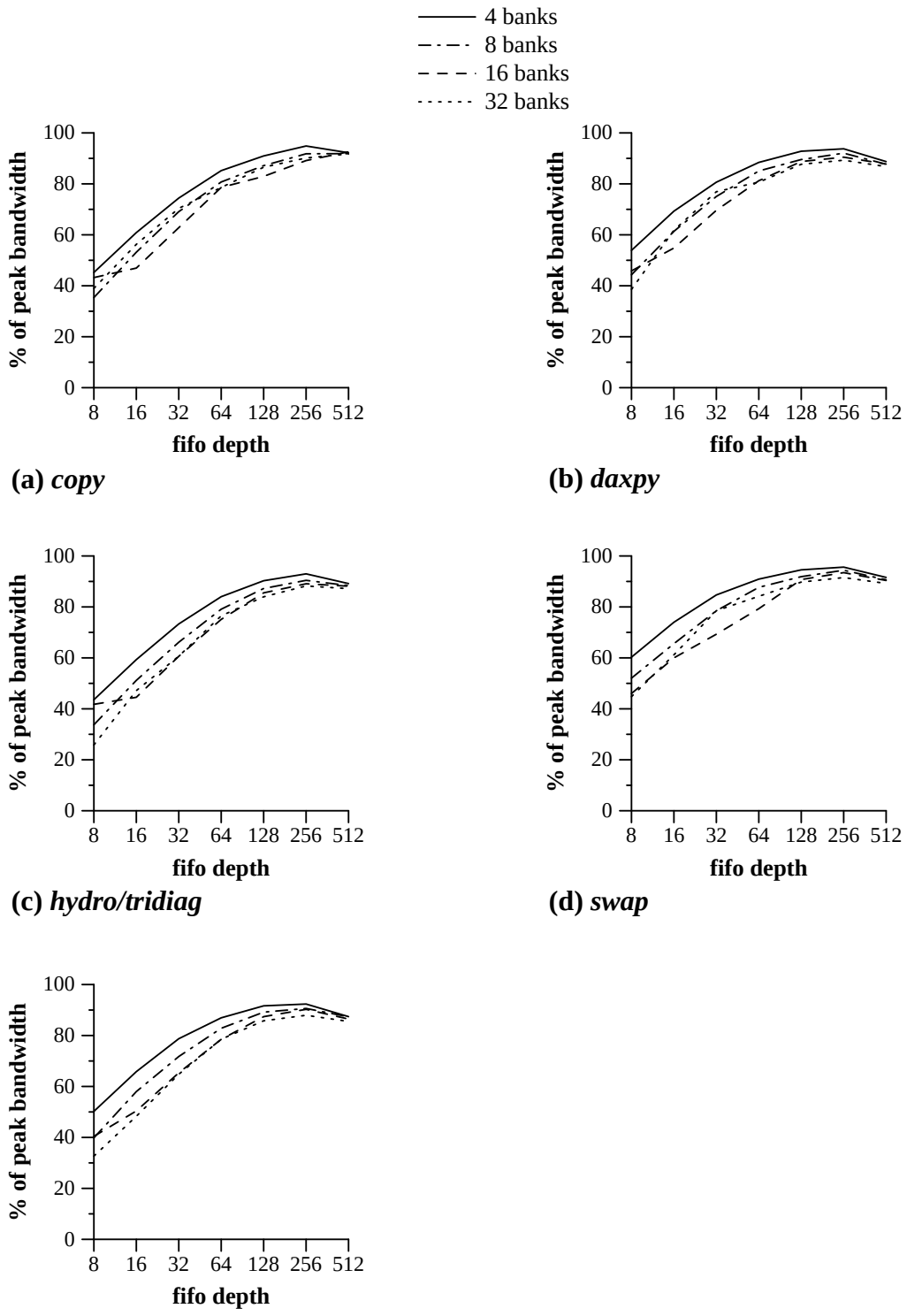



**(d)** ***swap***




**(e)** ***vaxpy***

**Figure 55 Prescheduled Token BC Performance for 2 CEs (Sequential FIFO Ordering, Staggered Alignment)**




**(a)** ***copy***

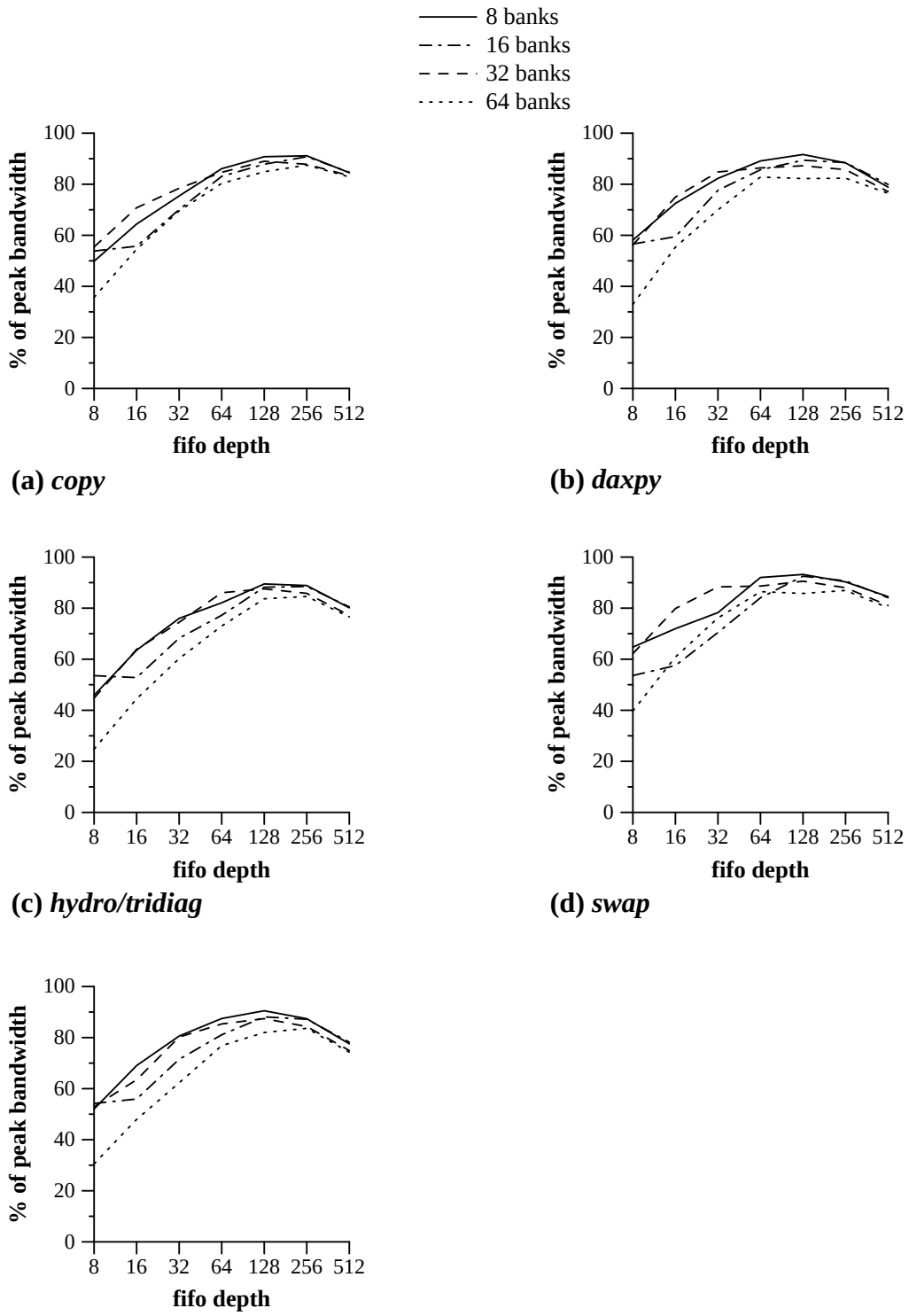
**(b)** ***daxpy***

**(c)** ***hydro/tridiag***

**(d)** ***swap***

**(e)** ***vaxpy***

**Figure 56 Prescheduled Token BC Performance for 4 CEs (Sequential FIFO Ordering, Staggered Alignment)**

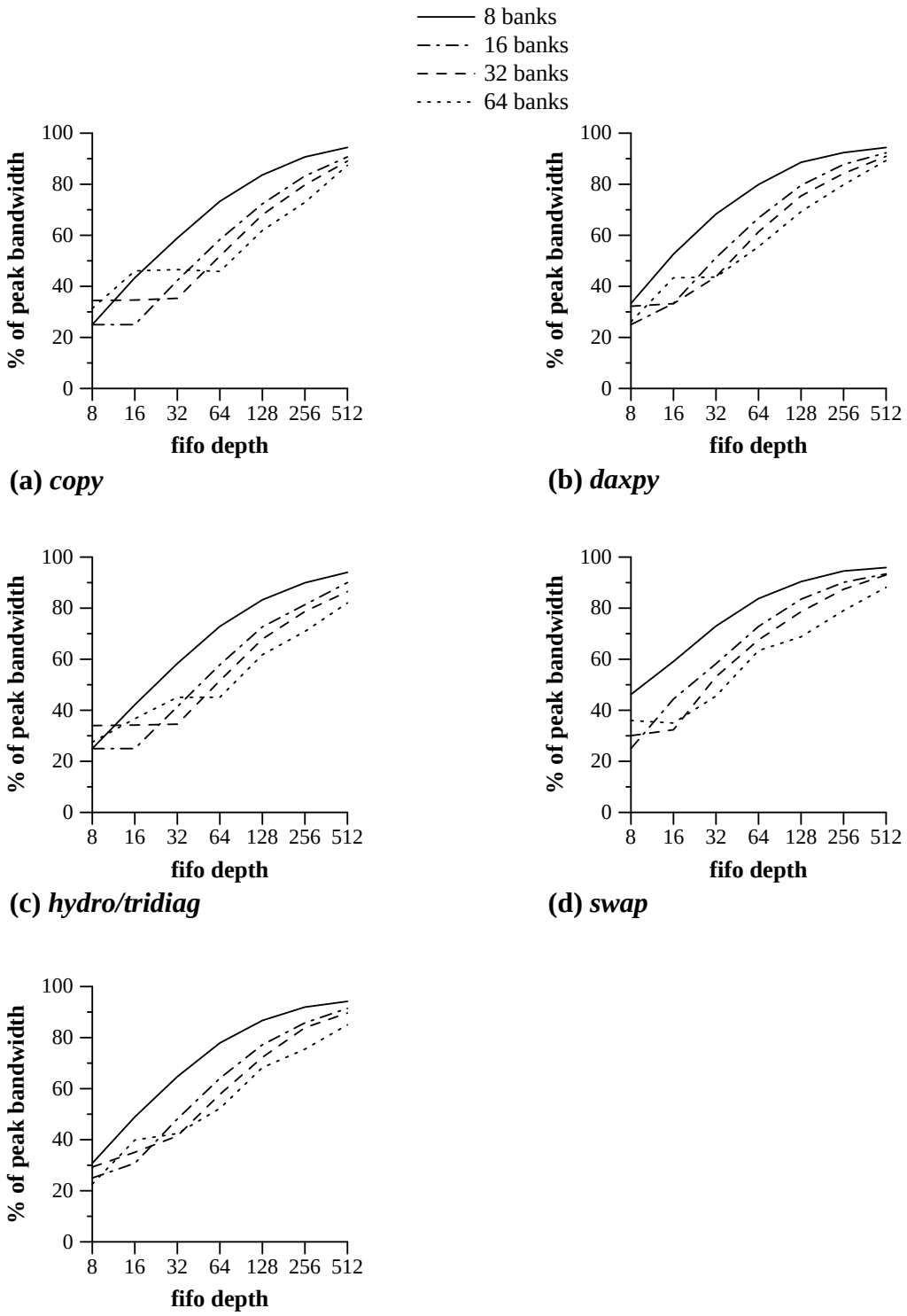



**(a) *copy*** **(b) *daxpy***

**(c) *hydro/tridiag*** **(d) *swap***

**(e) *vaxpy***

**Figure 57 Prescheduled Token BC Performance for 8 CEs (Sequential FIFO Ordering, Staggered Alignment)**




**(a) *copy*** **(b) *daxpy***

**(c) *hydro/tridiag*** **(d) *swap***

**(e) *vaxpy***

**Figure 58 Prescheduled Token BC Performance for 8 CEs (Sequential FIFO Ordering, Longer Vectors, Staggered Alignment)**

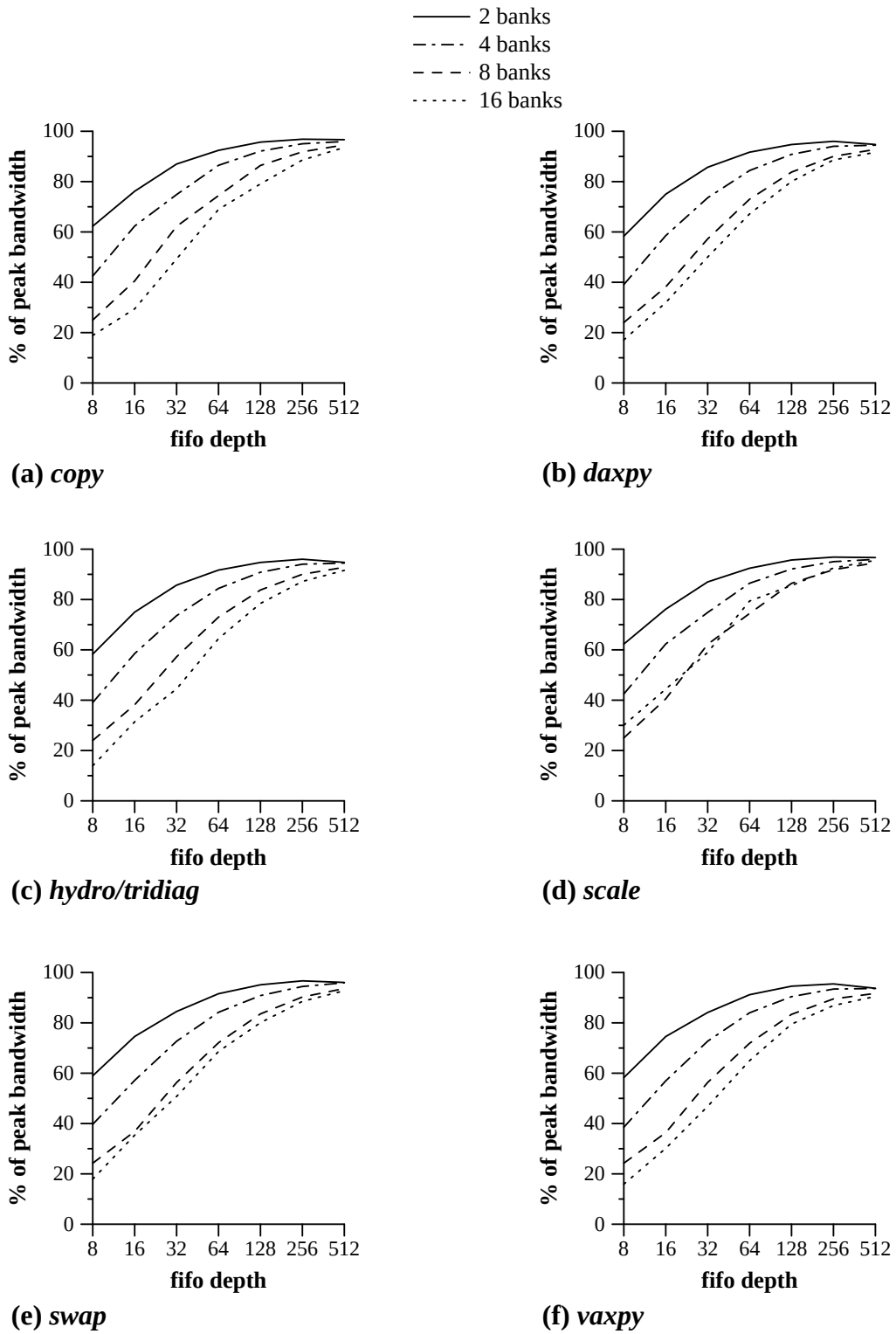



**Figure 59 Prescheduled FC Performance for 2 CEs**

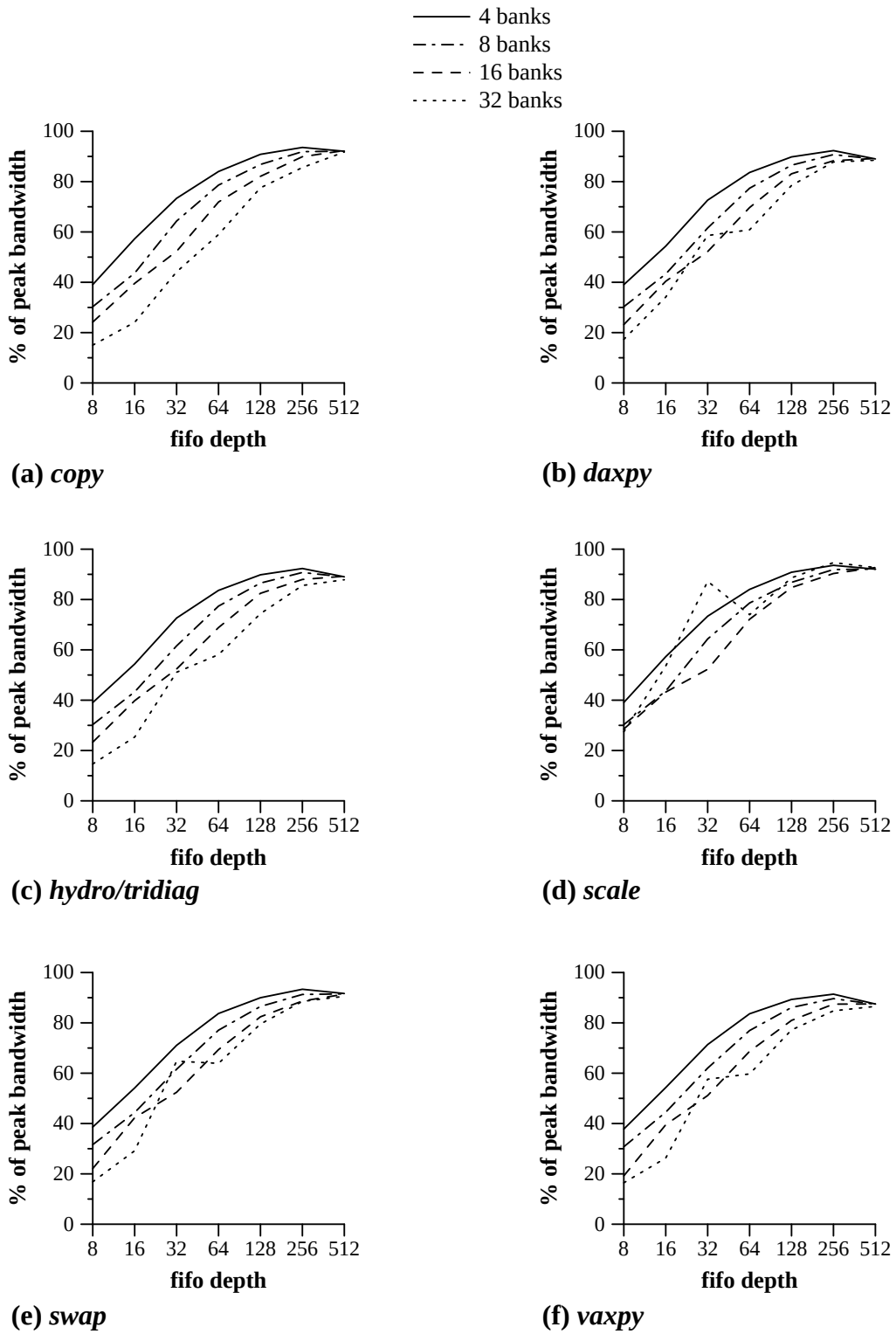



**Figure 60 Prescheduled FC Performance for 4 CEs**

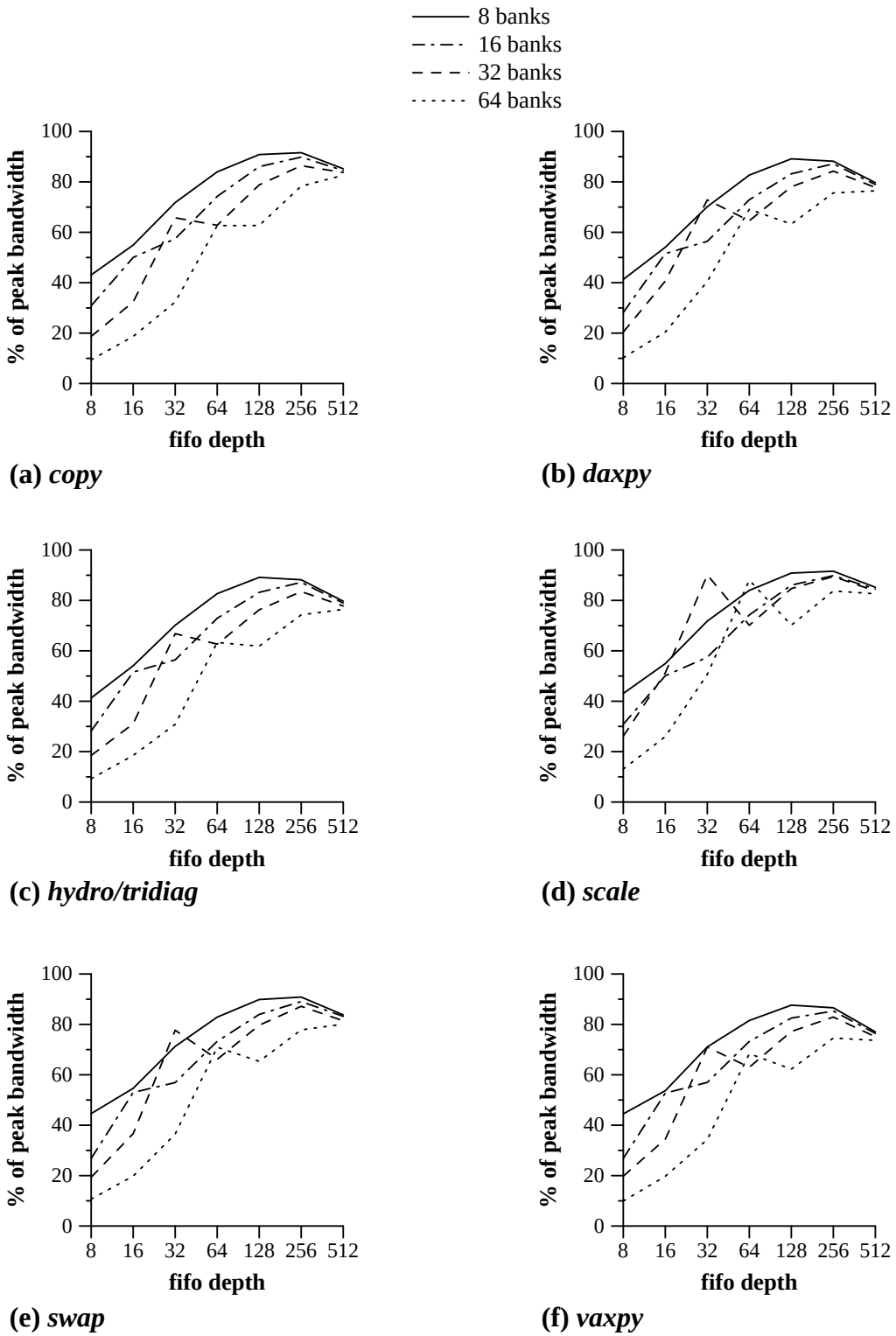



**Figure 61 Prescheduled FC Performance for 8 CEs**

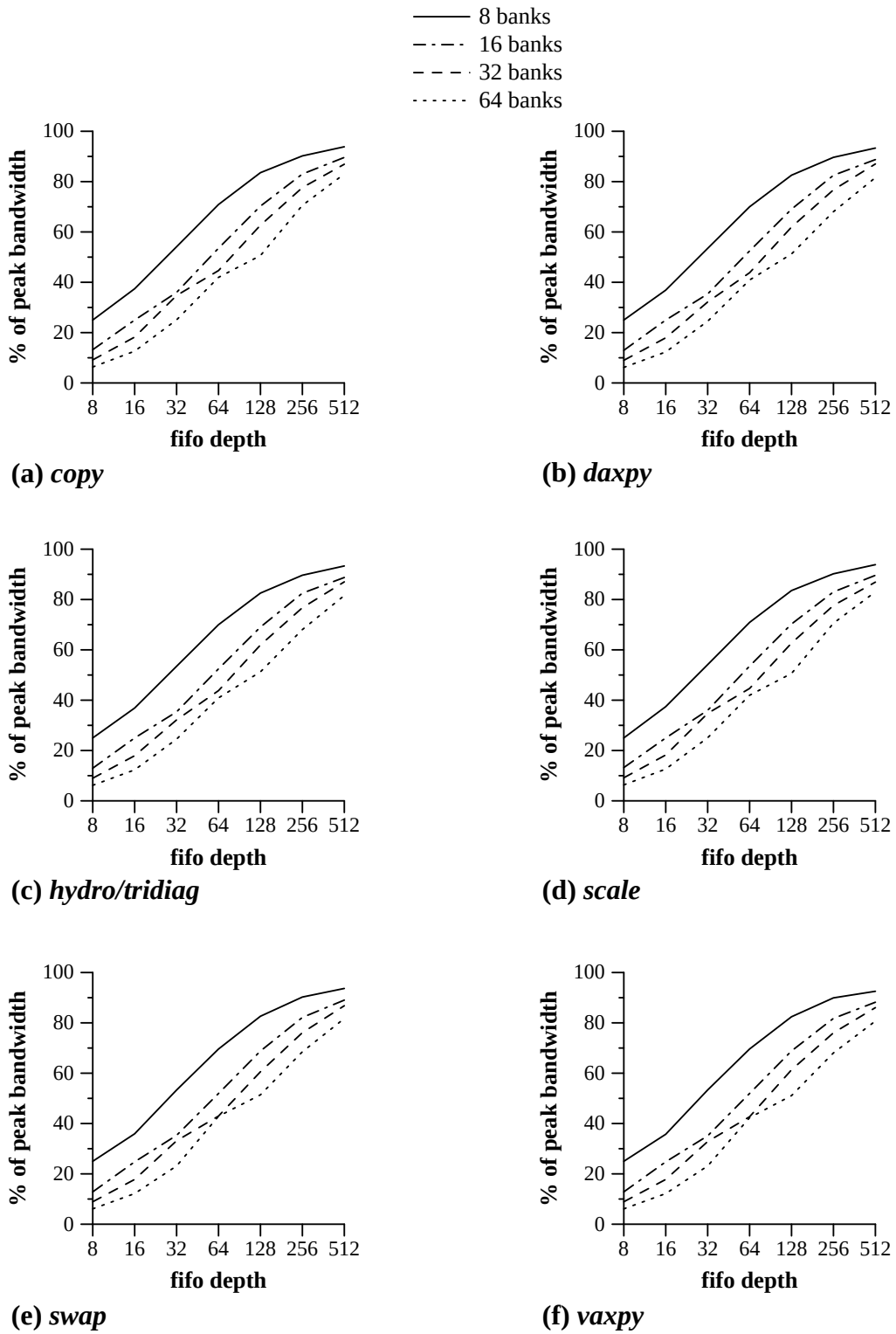



**Figure 62 Prescheduled FC Performance for 8 CEs (Longer Vectors)**

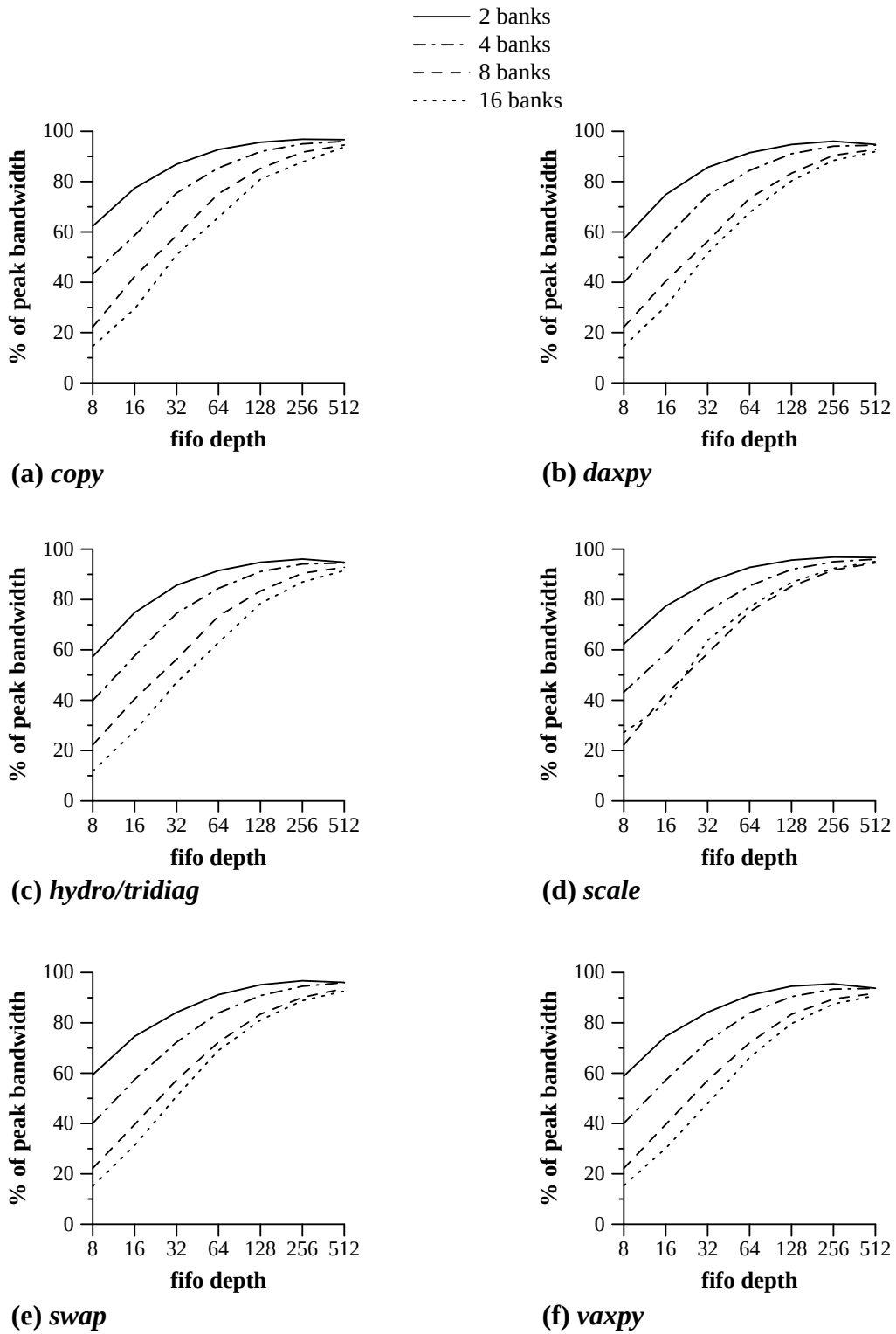



**Figure 63 Prescheduled FC Performance for 2 CEs (Staggered Alignment)**

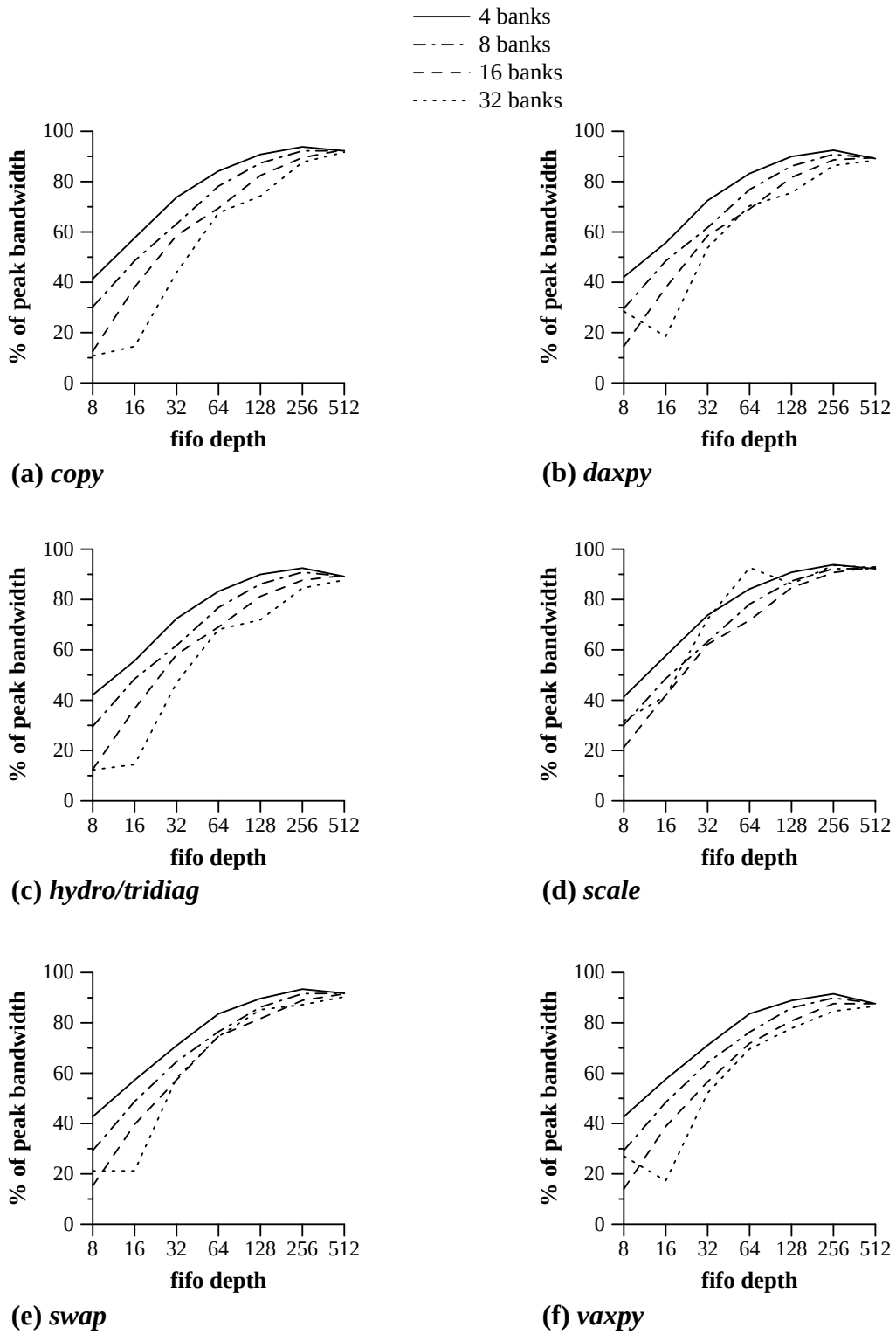



**Figure 64 Prescheduled FC Performance for 4 CEs (Staggered Alignment)**

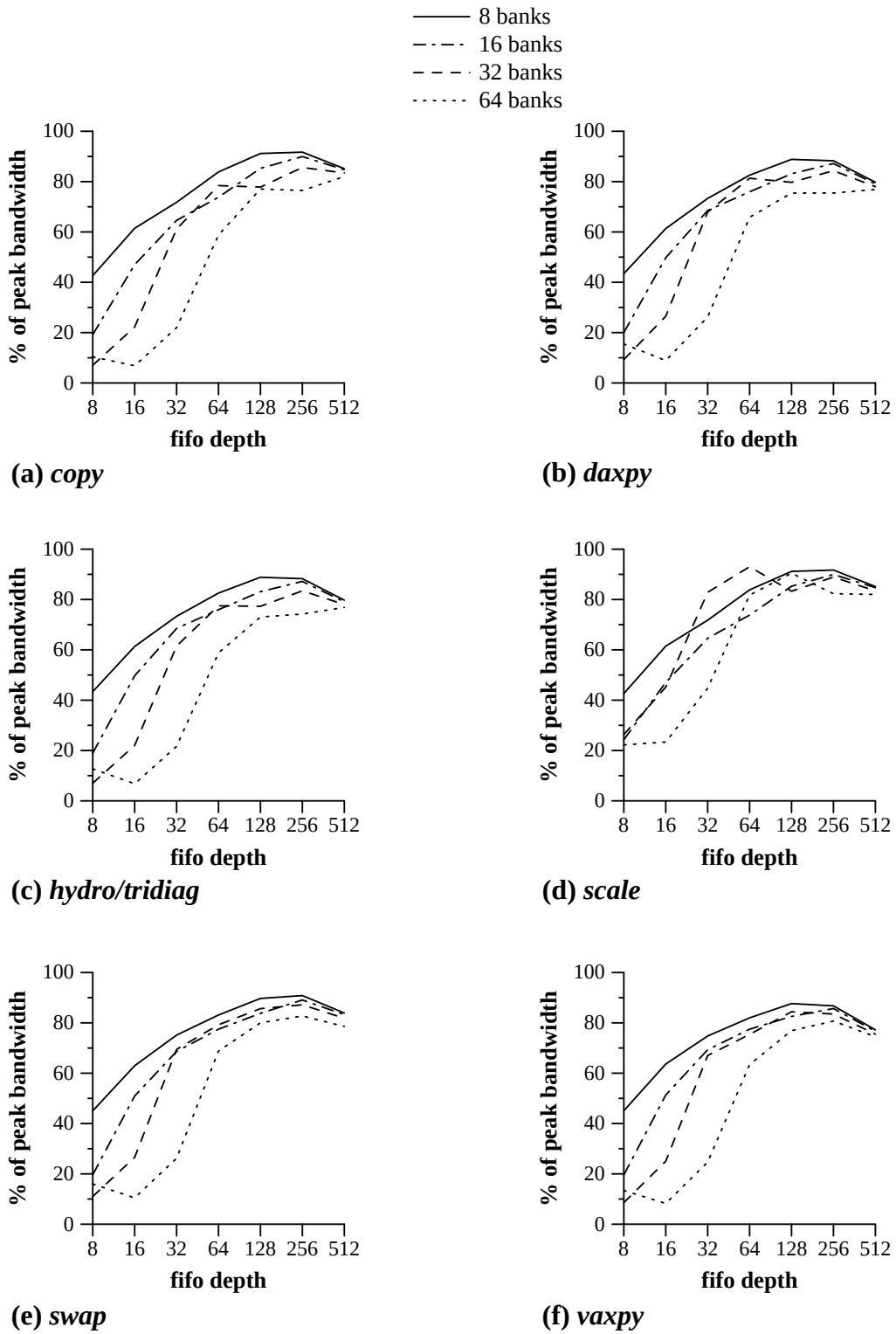



**Figure 65 Prescheduled FC Performance for 8 CEs (Staggered Alignment)**

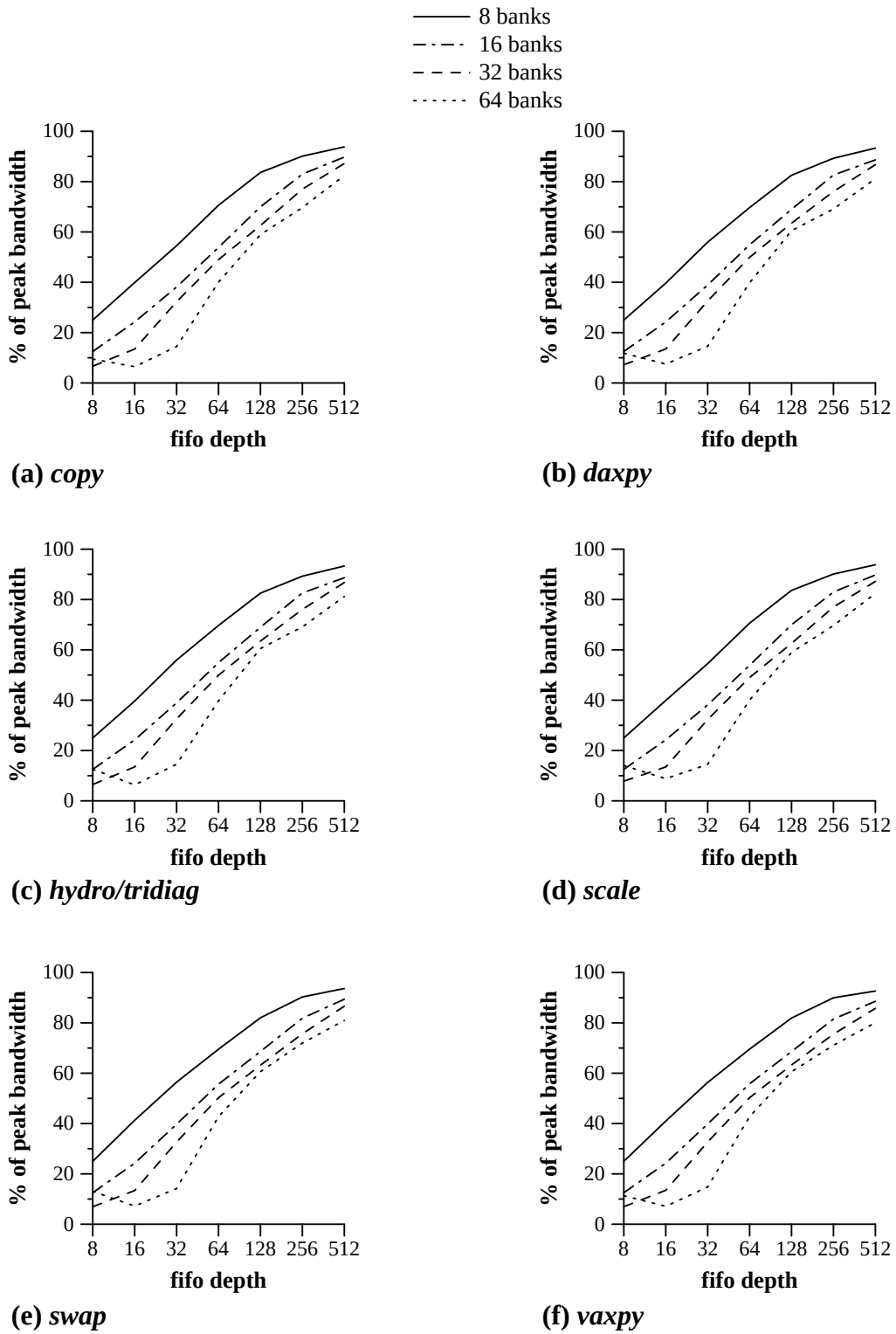



**Figure 66 Prescheduled FC Performance for 8 CEs (Longer Vectors, Staggered Alignment)**

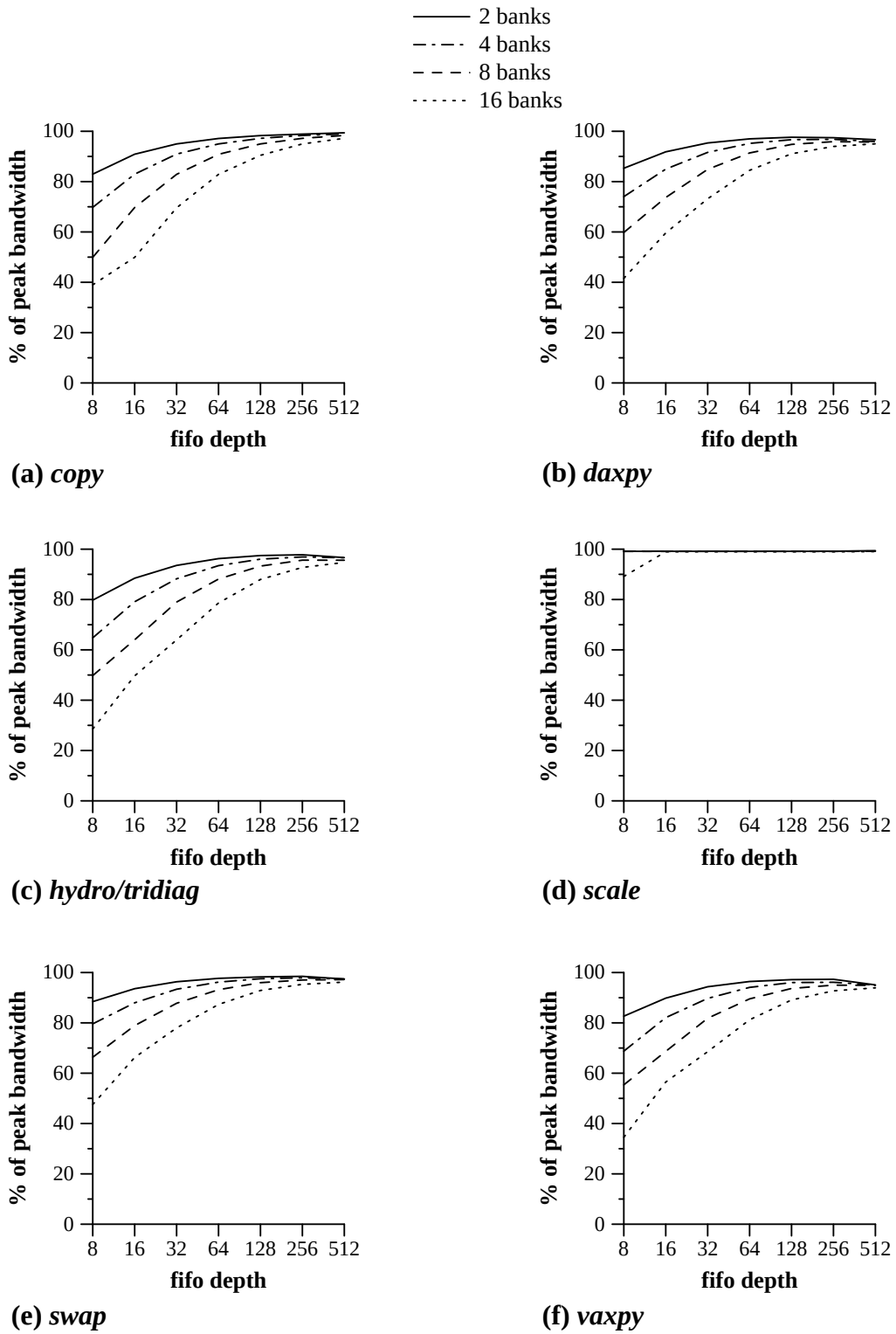



**Figure 67 Static Exhaustive BC Performance for 2 CEs**

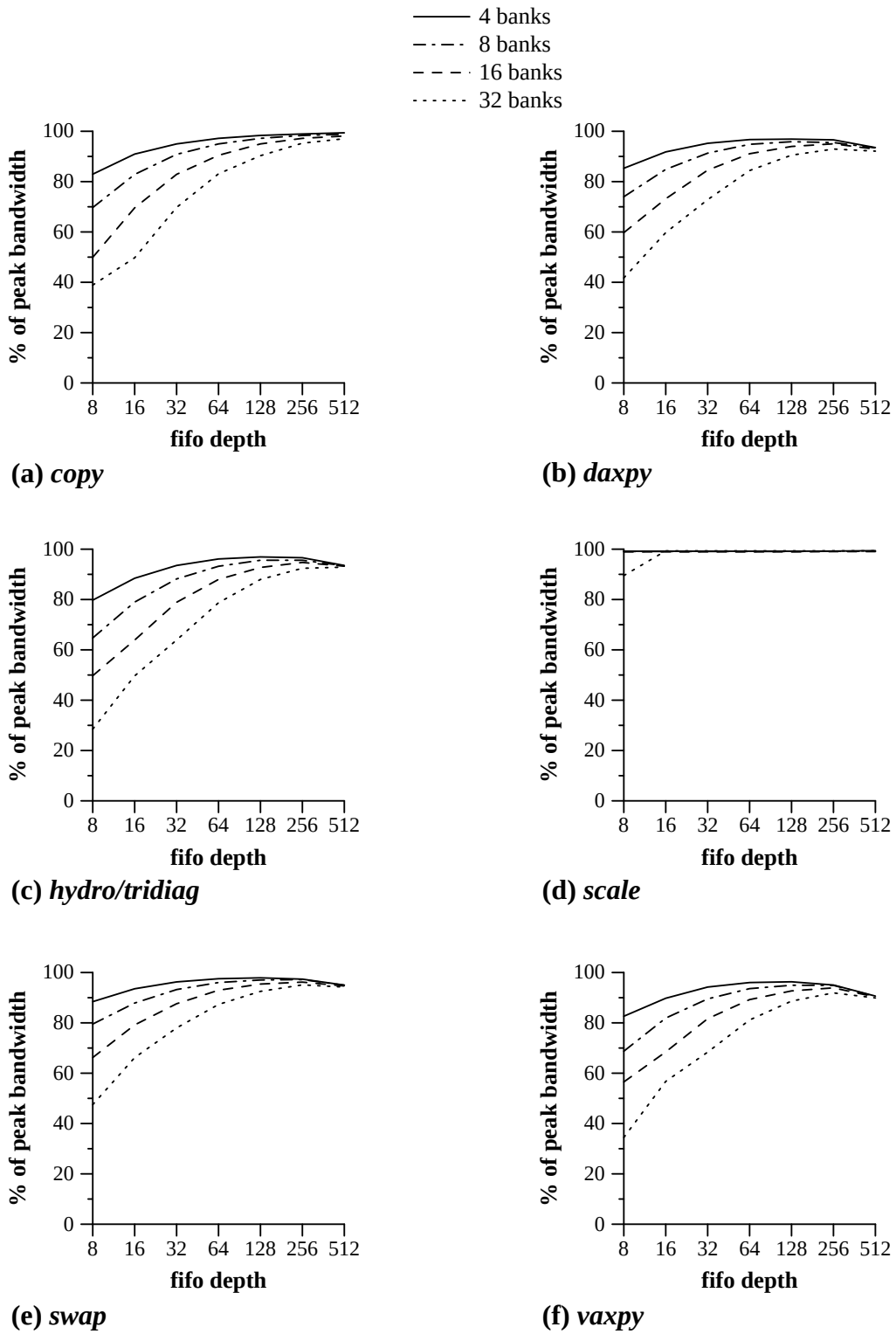



**Figure 68 Static Exhaustive BC Performance for 4 CEs**

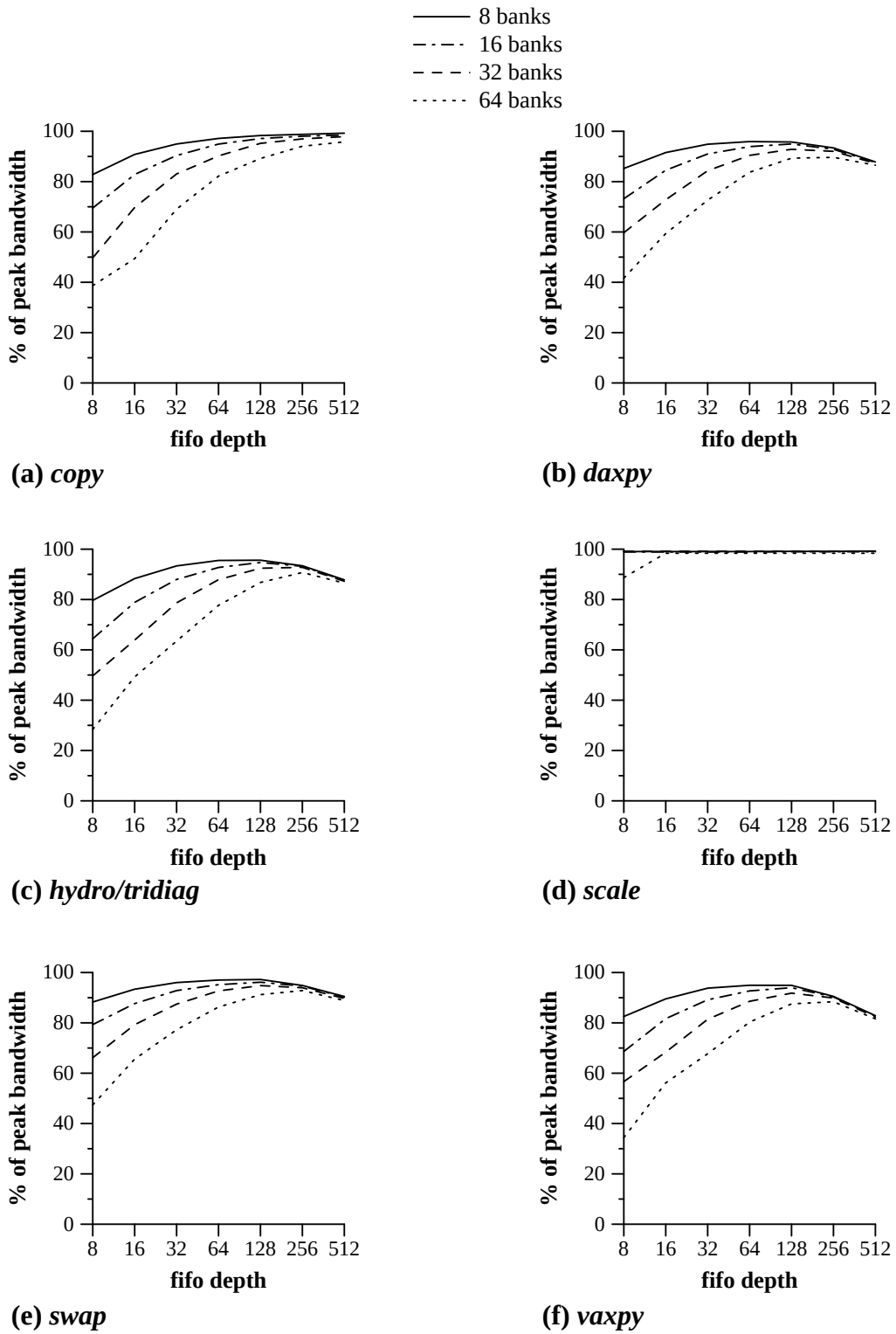
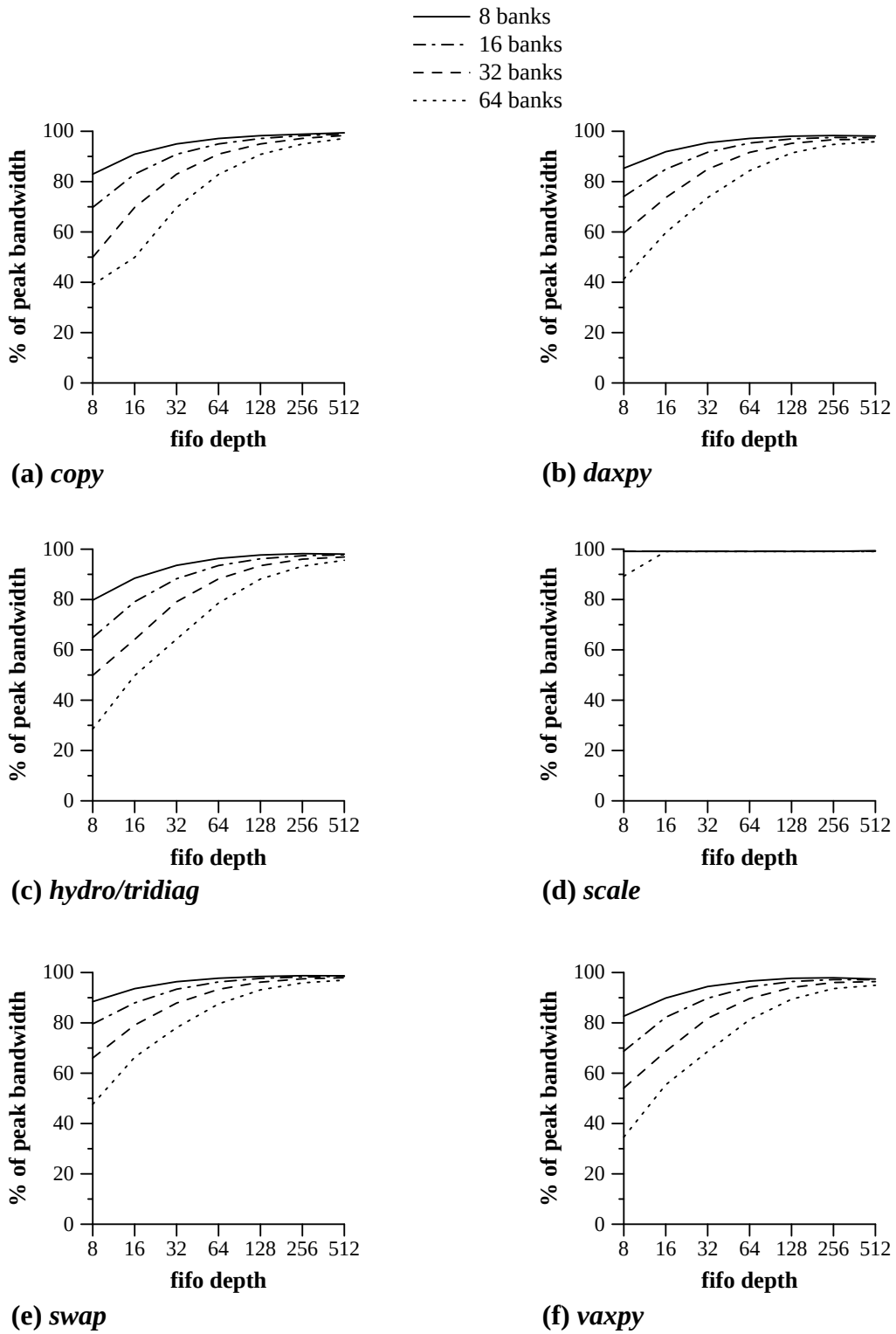



**Figure 69 Static Exhaustive BC Performance for 8 CEs**




**(a)** ***copy*** **(b)** ***daxpy***

**(c)** ***hydro/tridiag*** **(d)** ***scale***

**(e)** ***swap*** **(f)** ***vaxpy***

**Figure 70 Static Exhaustive BC Performance for 8 CEs (Longer Vectors)**

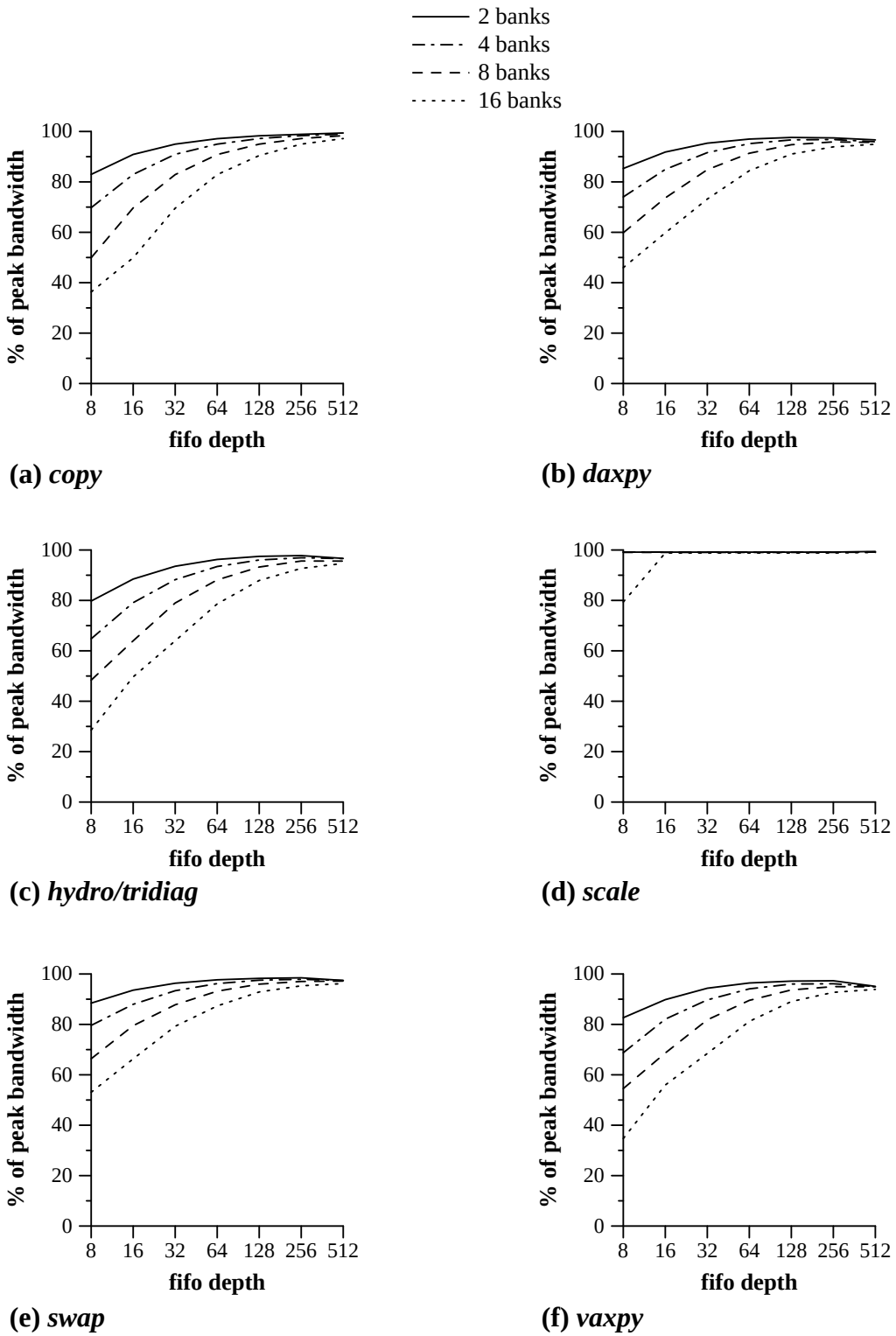



**Figure 71 Static Token BC Performance for 2 CEs**

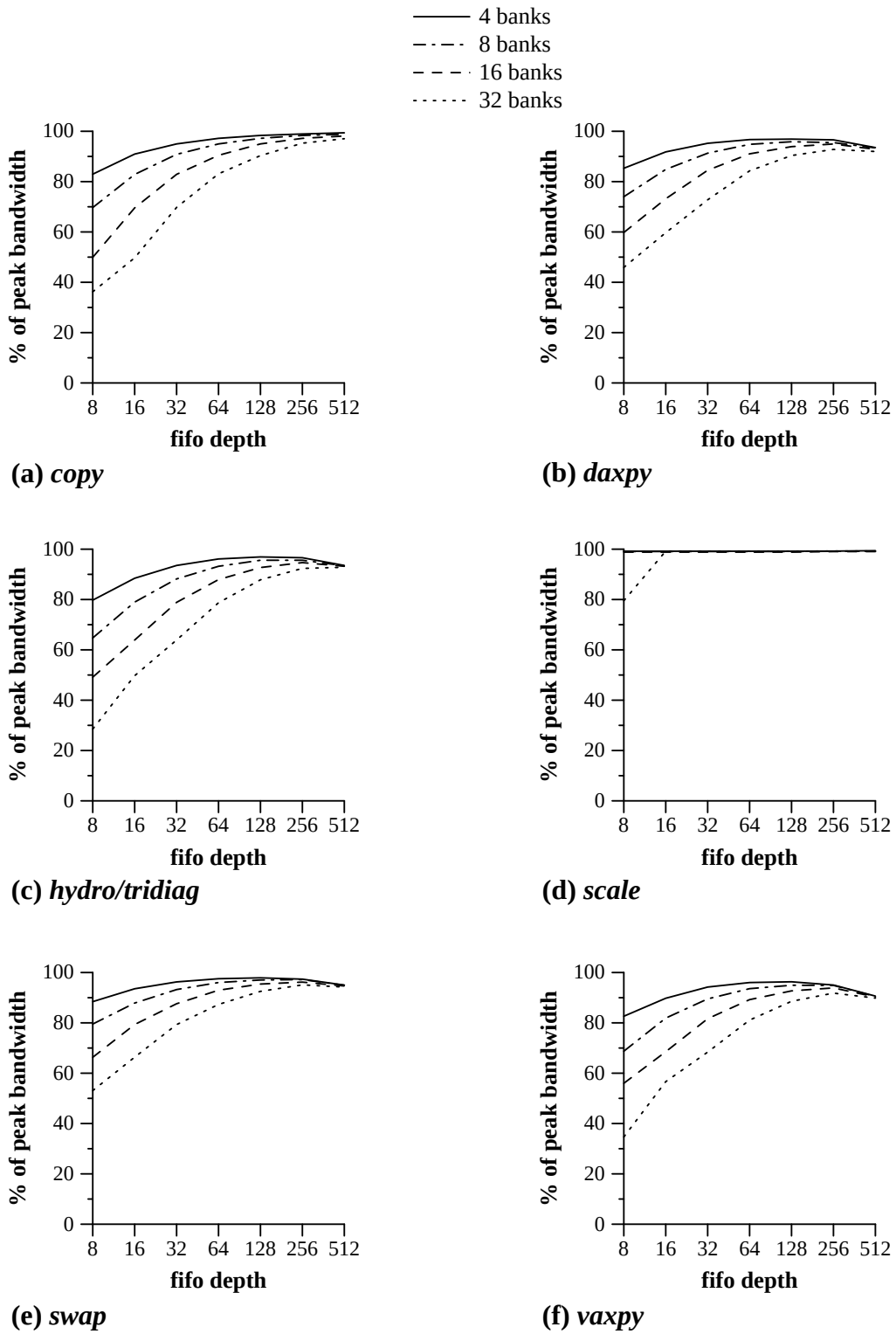



**Figure 72 Static Token BC Performance for 4 CEs**

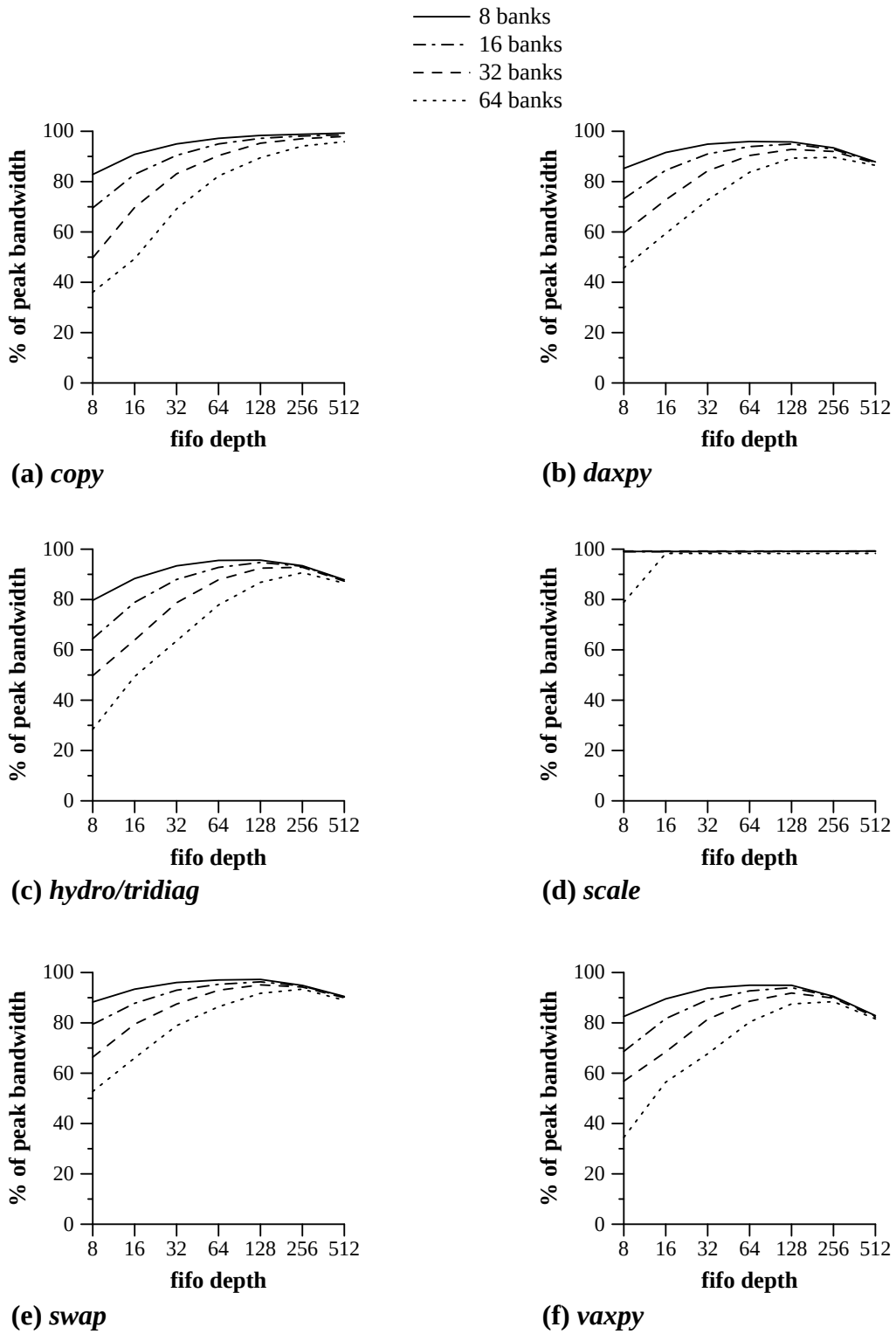



**Figure 73 Static Token BC Performance for 8 CEs**

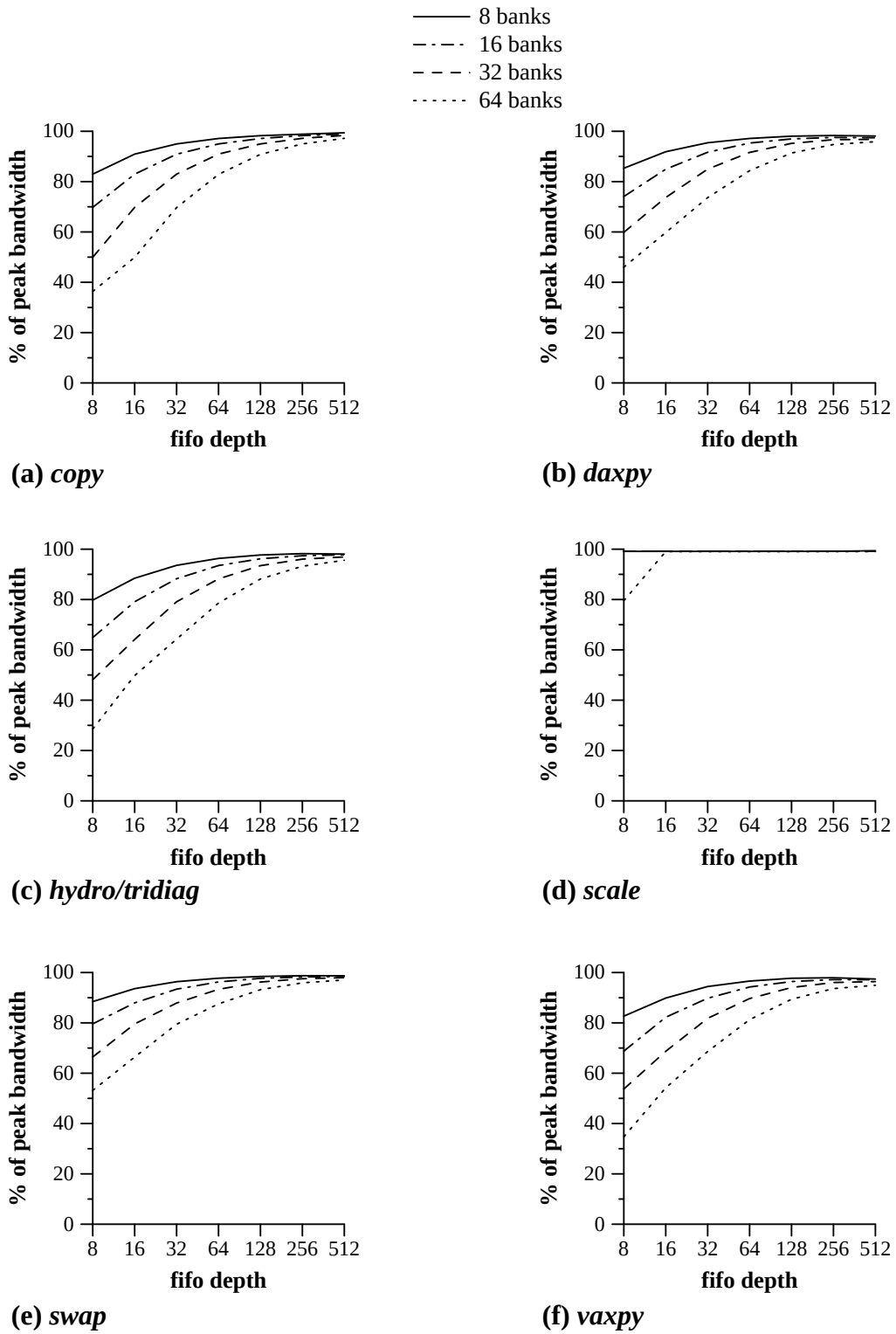
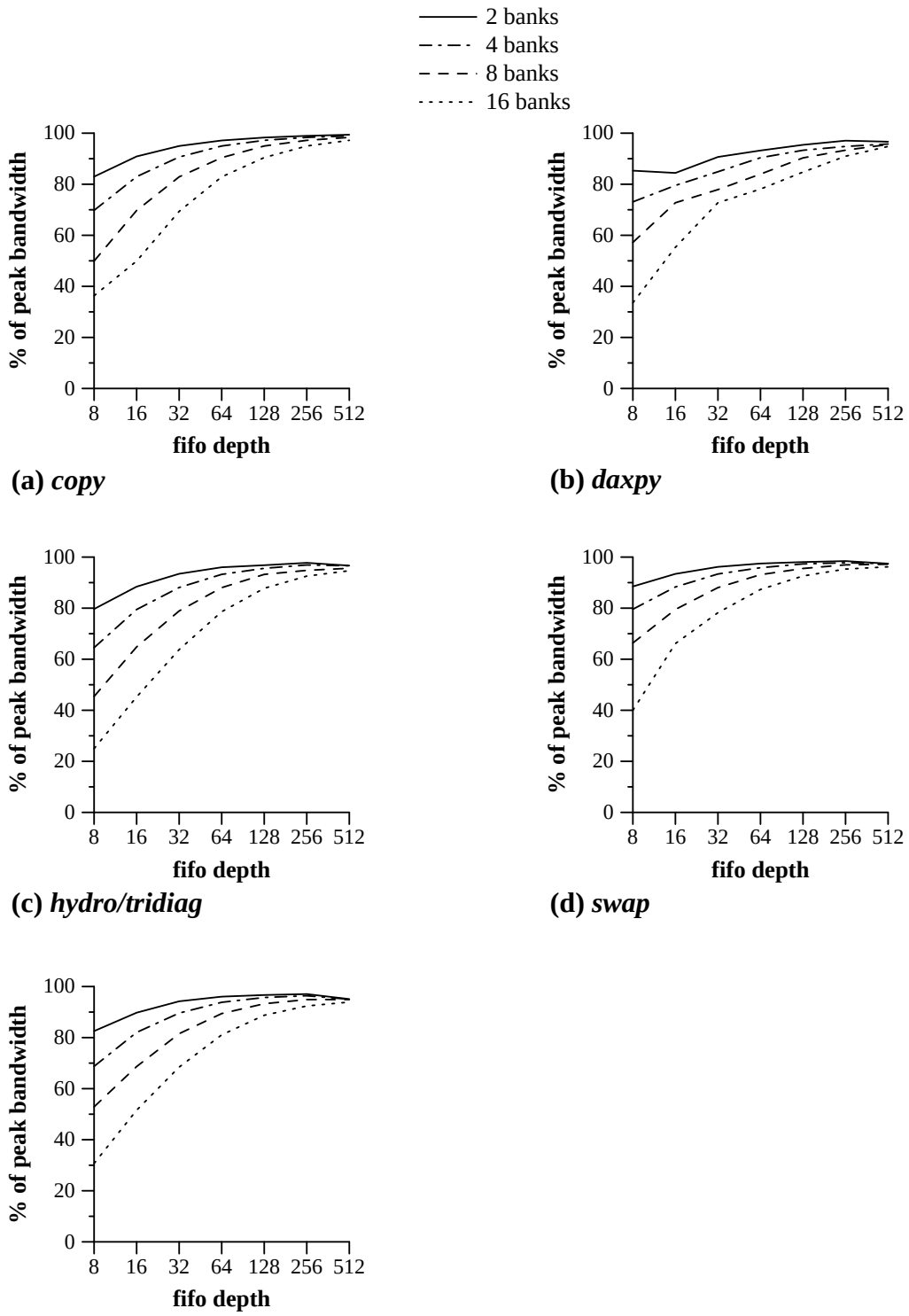



**Figure 74 Static Token BC Performance for 8 CEs (Longer Vectors)**

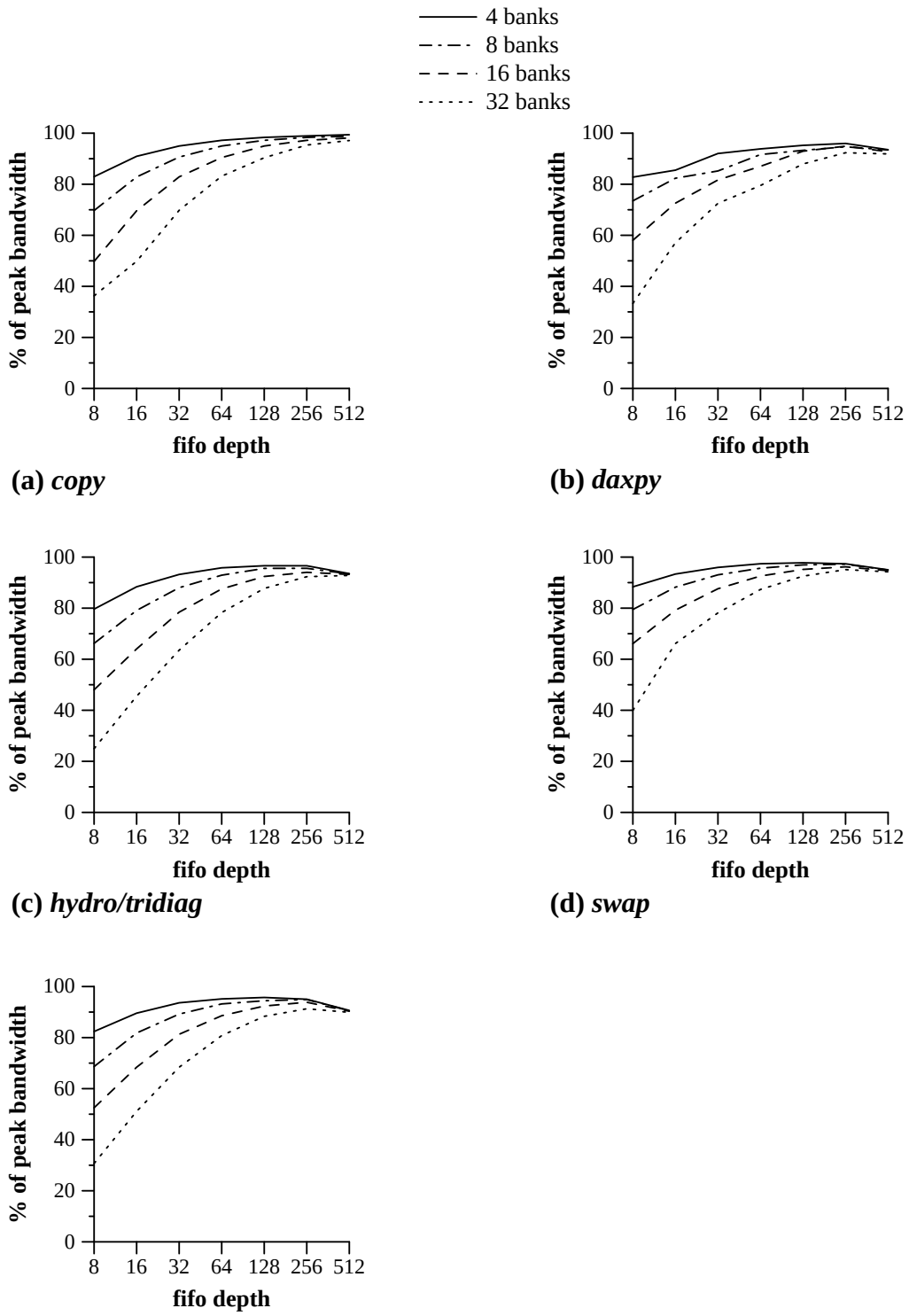



**(a) *copy*** **(b) *daxpy***

**(c) *hydro/tridiag*** **(d) *swap***

**(e) *vaxpy***

**Figure 75 Static Token BC Performance for 2 CEs (Staggered Alignment)**




**(a)** ***copy***

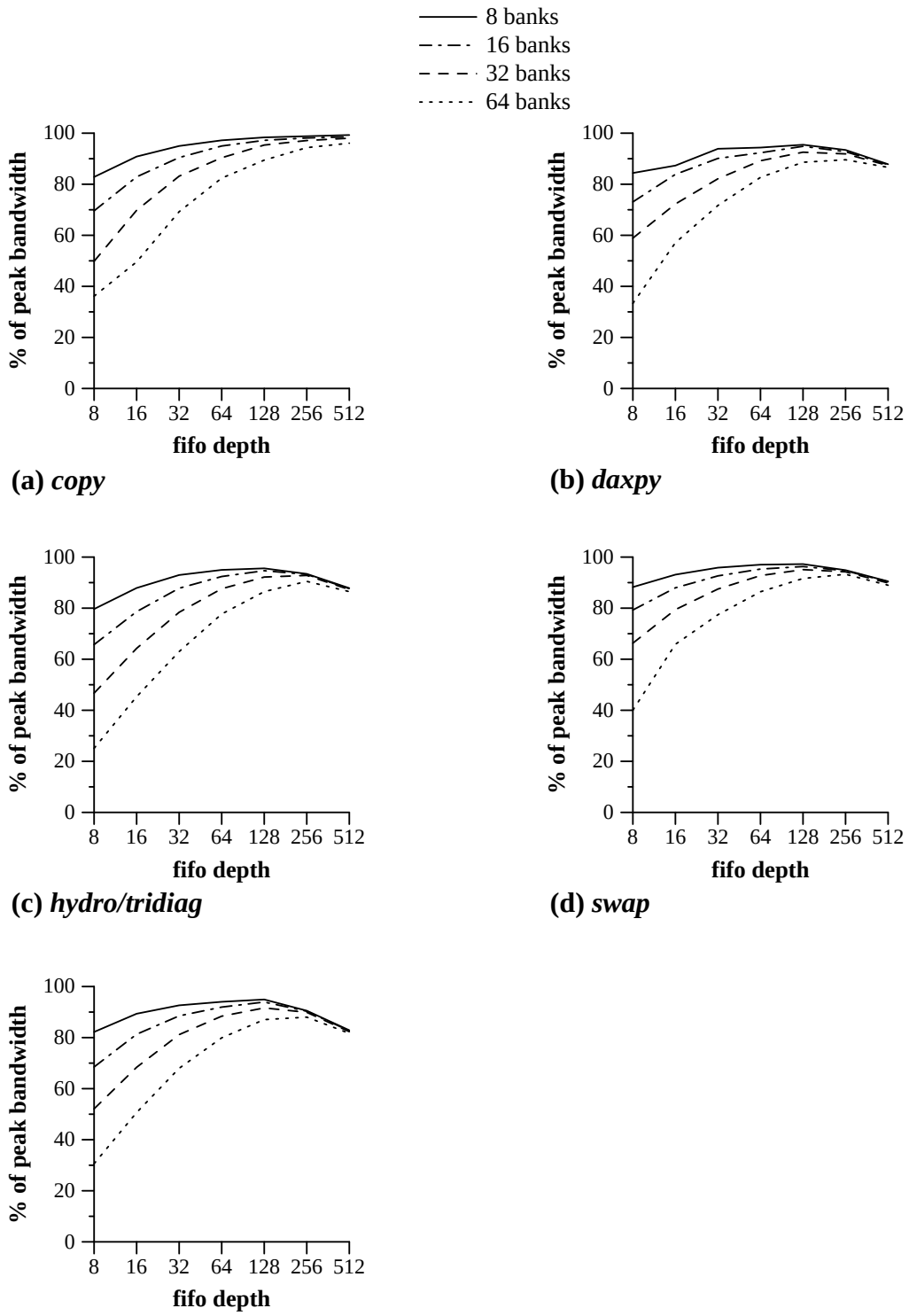
**(b)** ***daxpy***

**(c)** ***hydro/tridiag***

**(d)** ***swap***

**(e)** ***vaxpy***

**Figure 76 Static Token BC Performance for 4 CEs (Staggered Alignment)**




**(a) *copy*** **(b) *daxpy***

**(c) *hydro/tridiag*** **(d) *swap***

**(e) *vaxpy***

**Figure 77 Static Token BC Performance for 8 CEs (Staggered Alignment)**

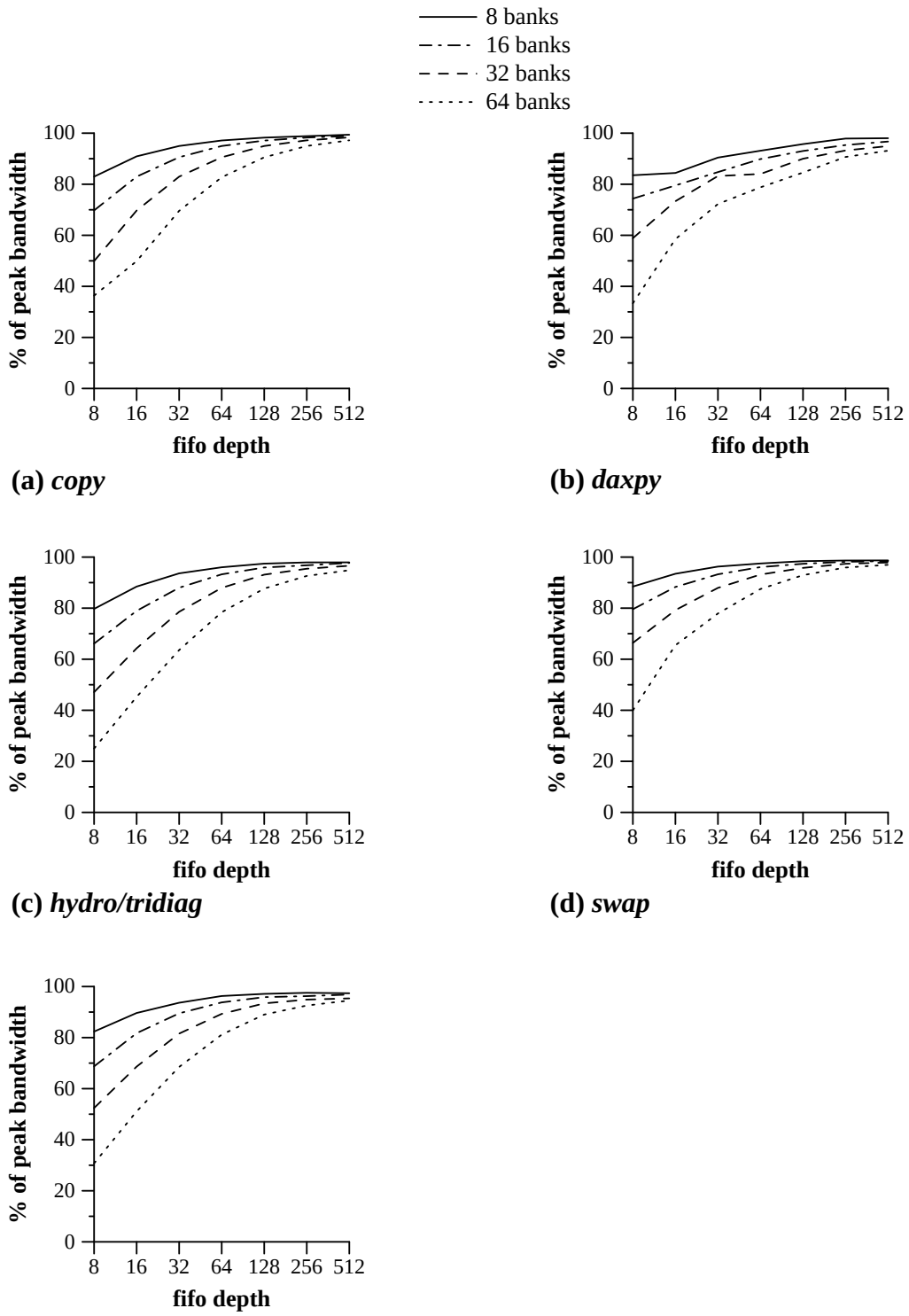
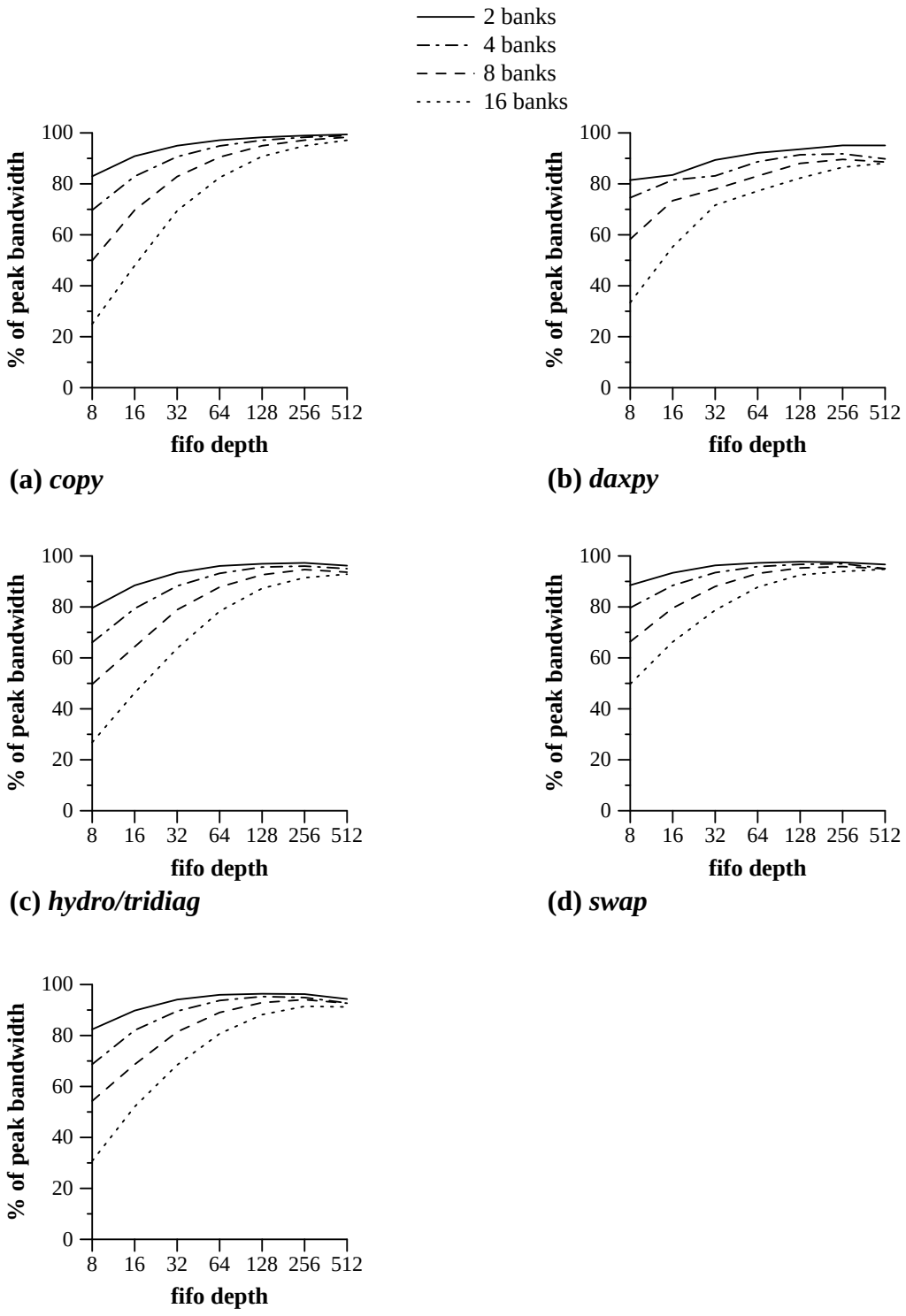



**Figure 78 Static Token BC Performance for 8 CEs (Longer Vectors, Staggered Alignment)**

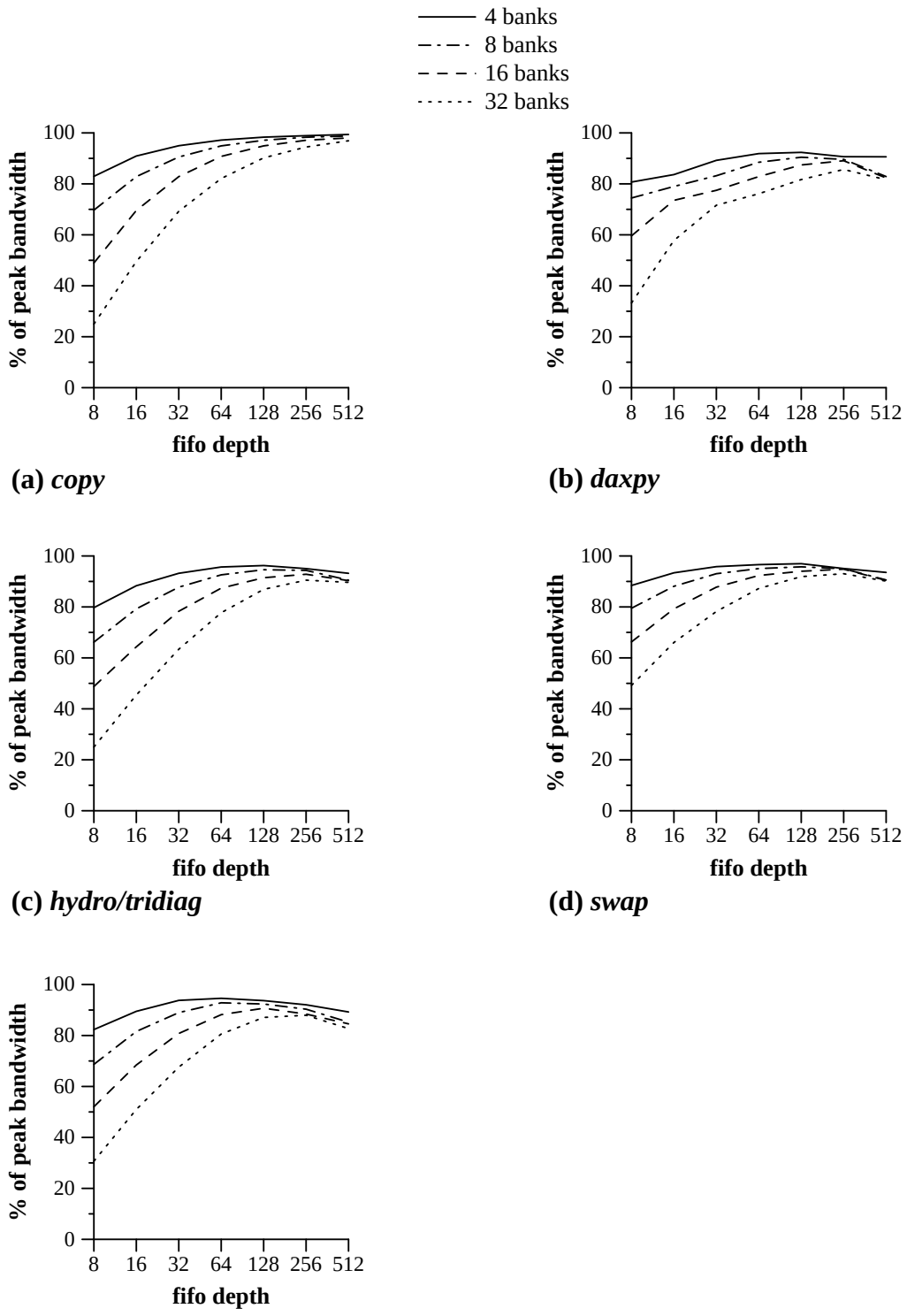



**(a) *copy*** **(b) *daxpy***

**(c) *hydro/tridiag*** **(d) *swap***

**(e) *vaxpy***

**Figure 79 Static Token BC Performance for 2 CEs (Sequential FIFO Ordering, Staggered Alignment)**




**(a) *copy*** **(b) *daxpy***

**(c) *hydro/tridiag*** **(d) *swap***

**(e) *vaxpy***

**Figure 80 Static Token BC Performance for 4 CEs (Sequential FIFO Ordering, Staggered Alignment)**

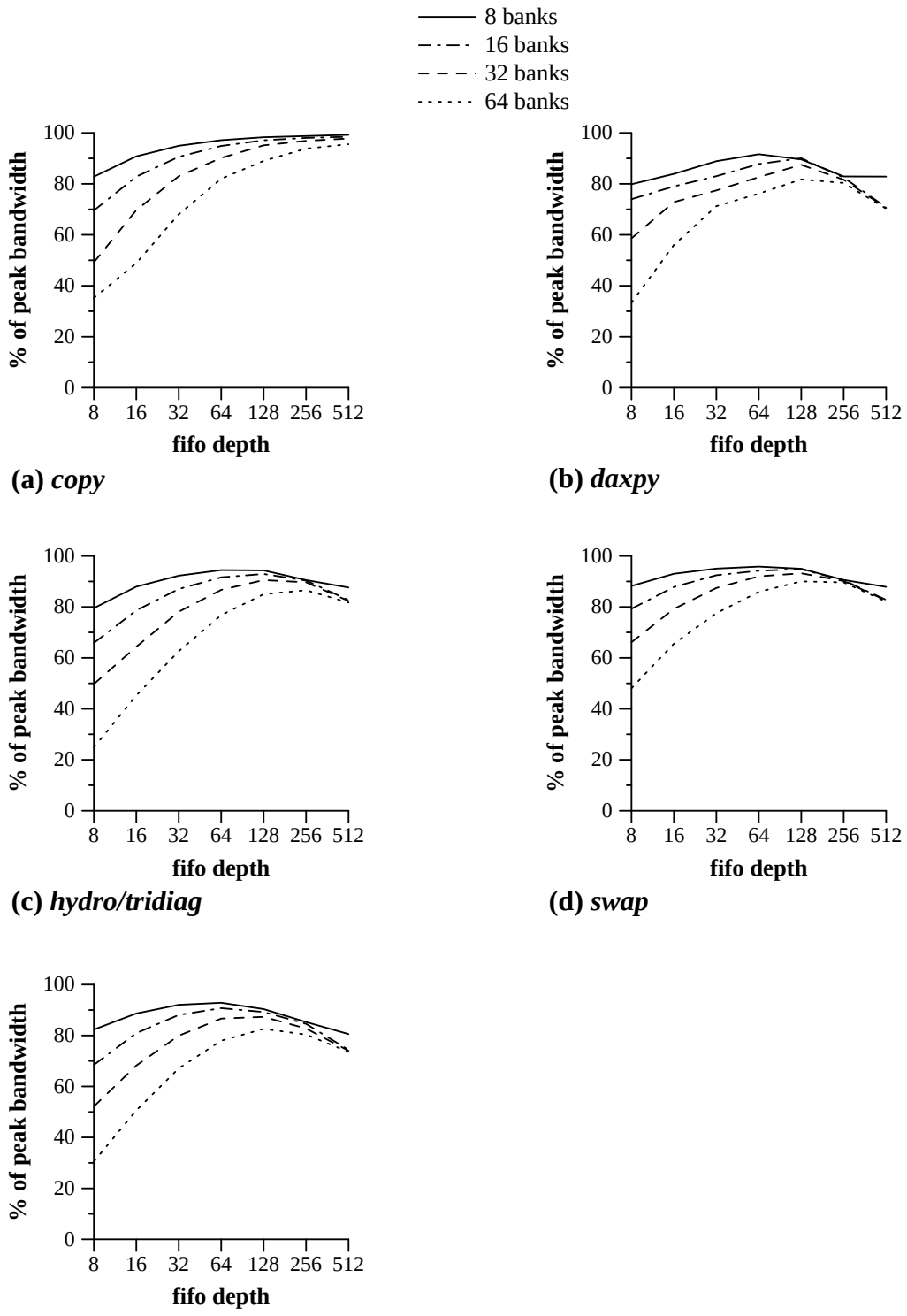
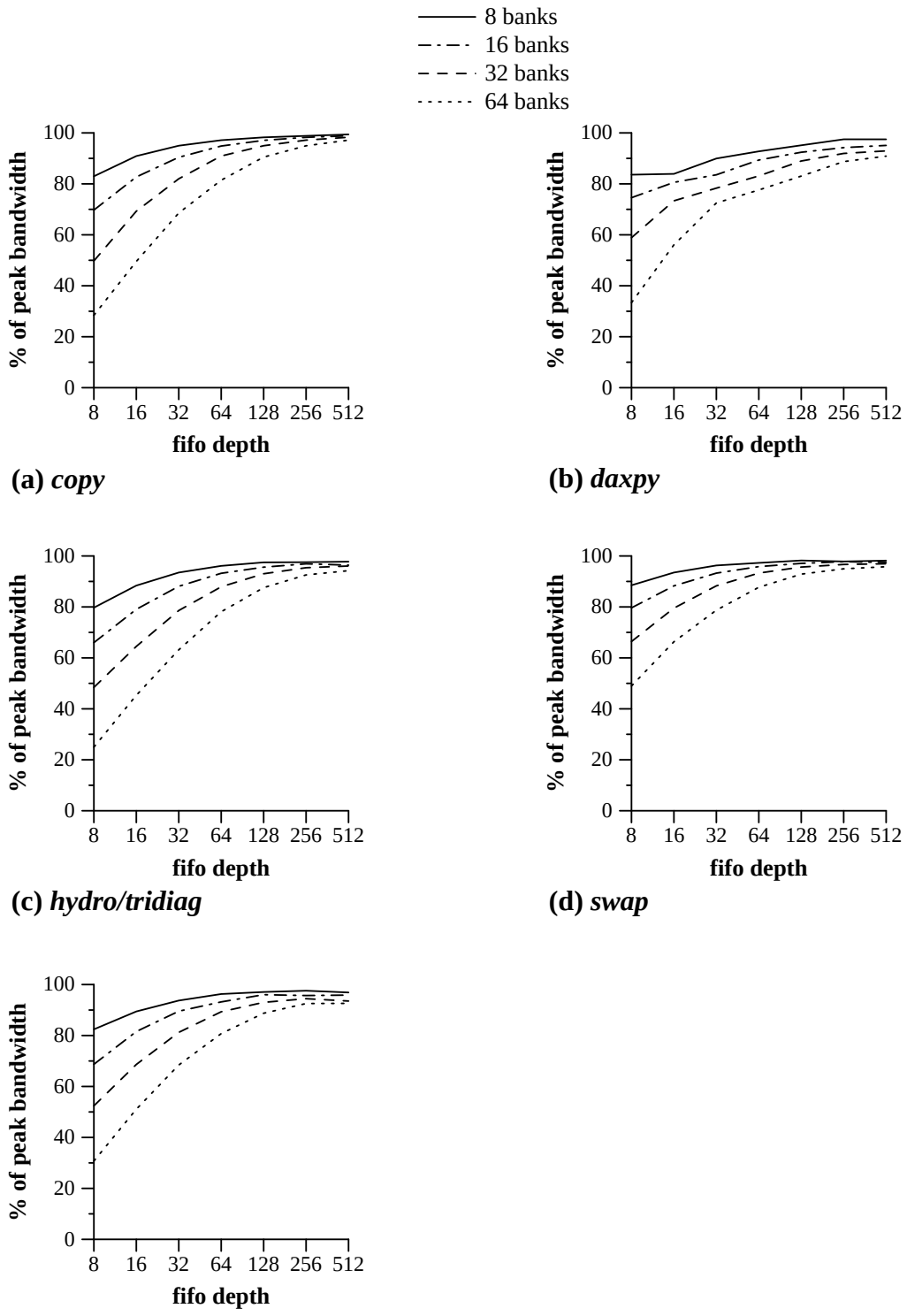



**Figure 81 Static Token BC Performance for 8 CEs (Sequential FIFO Ordering, Staggered Alignment)**




**(a) *copy*** **(b) *daxpy***

**(c) *hydro/tridiag*** **(d) *swap***

**(e) *vaxpy***

**Figure 82 Static Token BC Performance for 8 CEs**
**(Sequential FIFO Ordering, Longer Vectors, Staggered Alignment)**

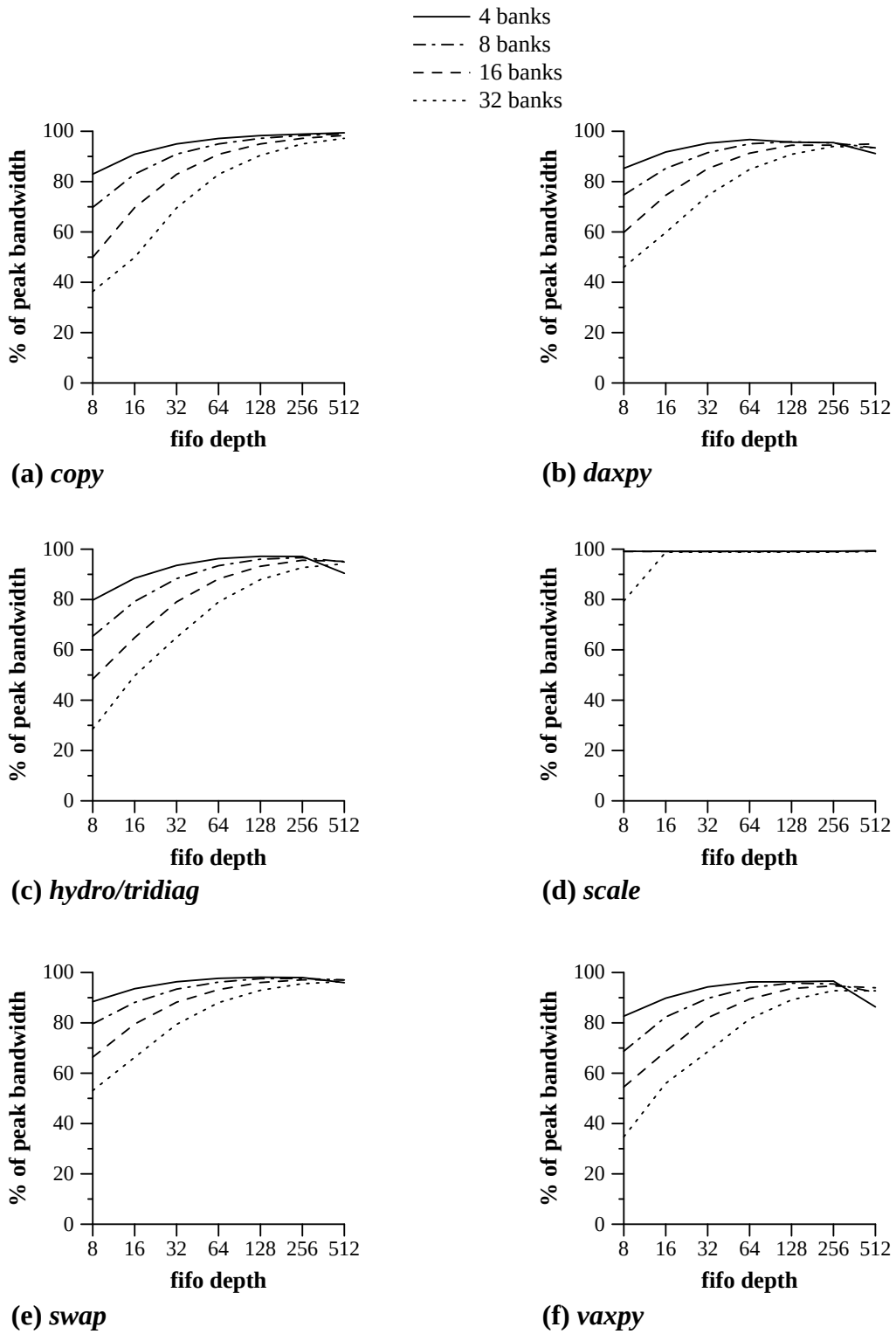



**Figure 83 Static Token TBC Performance for 2 CEs**

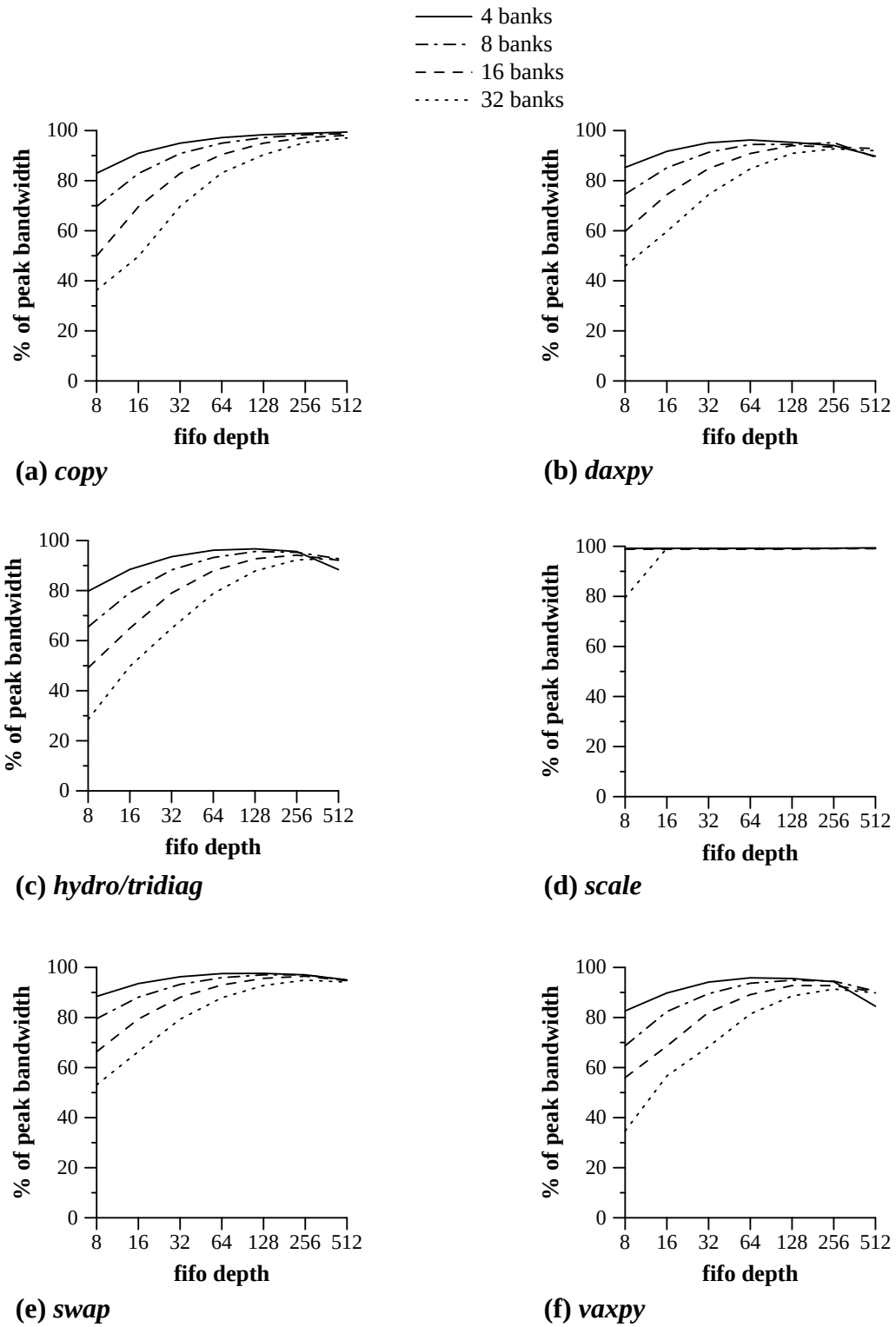



**Figure 84 Static Token TBC Performance for 4 CEs**

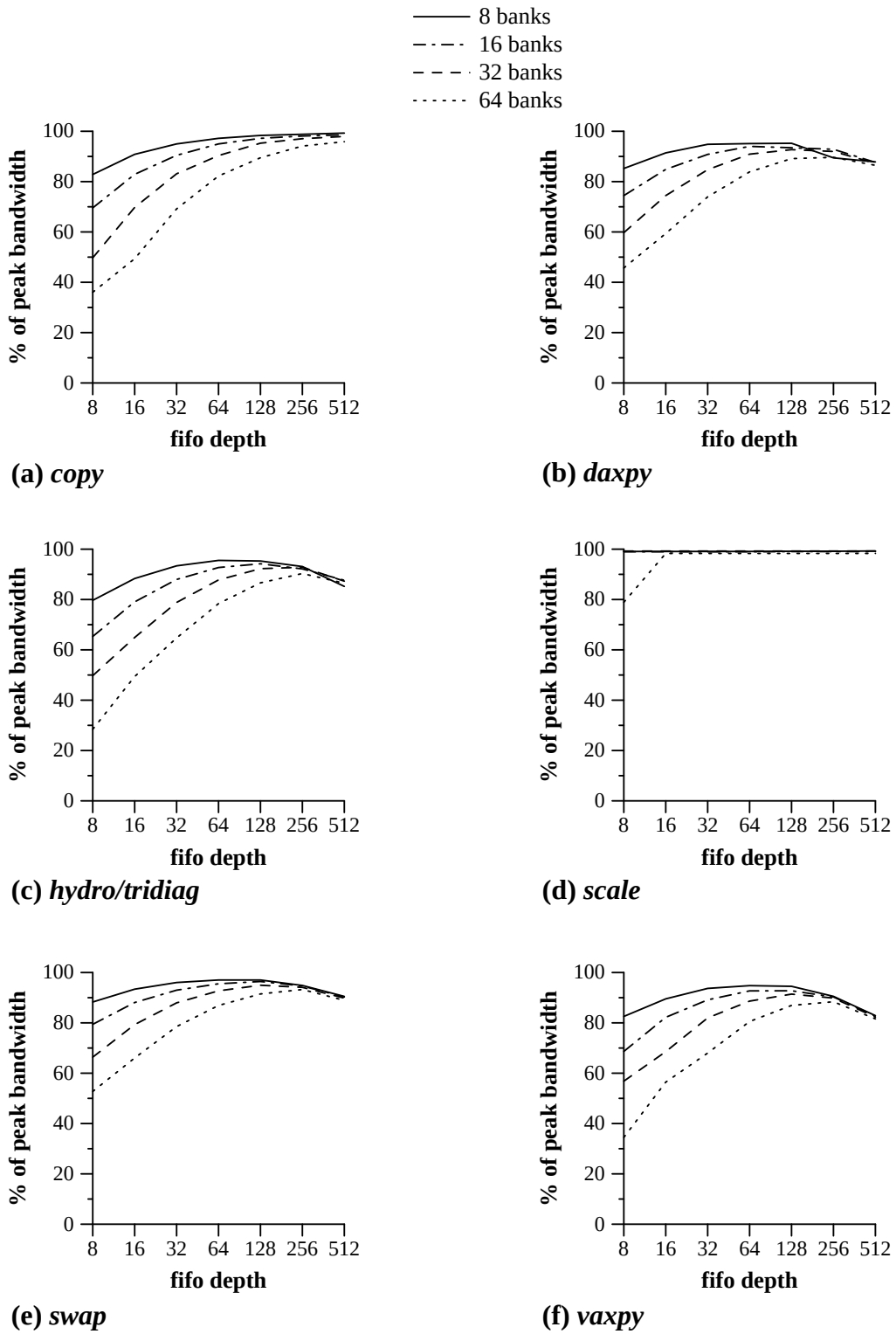



**Figure 85 Static Token TBC Performance for 8 CEs**

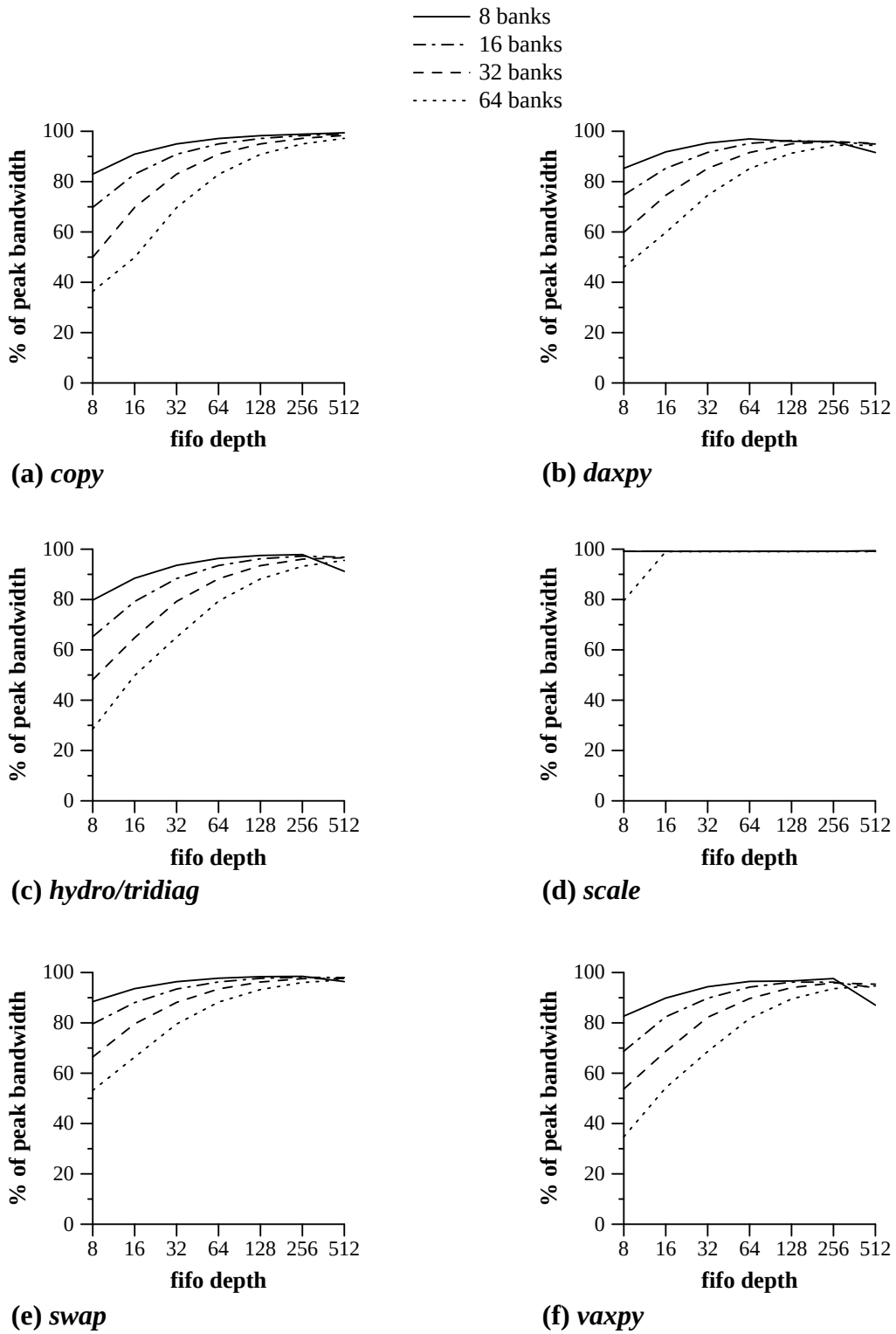
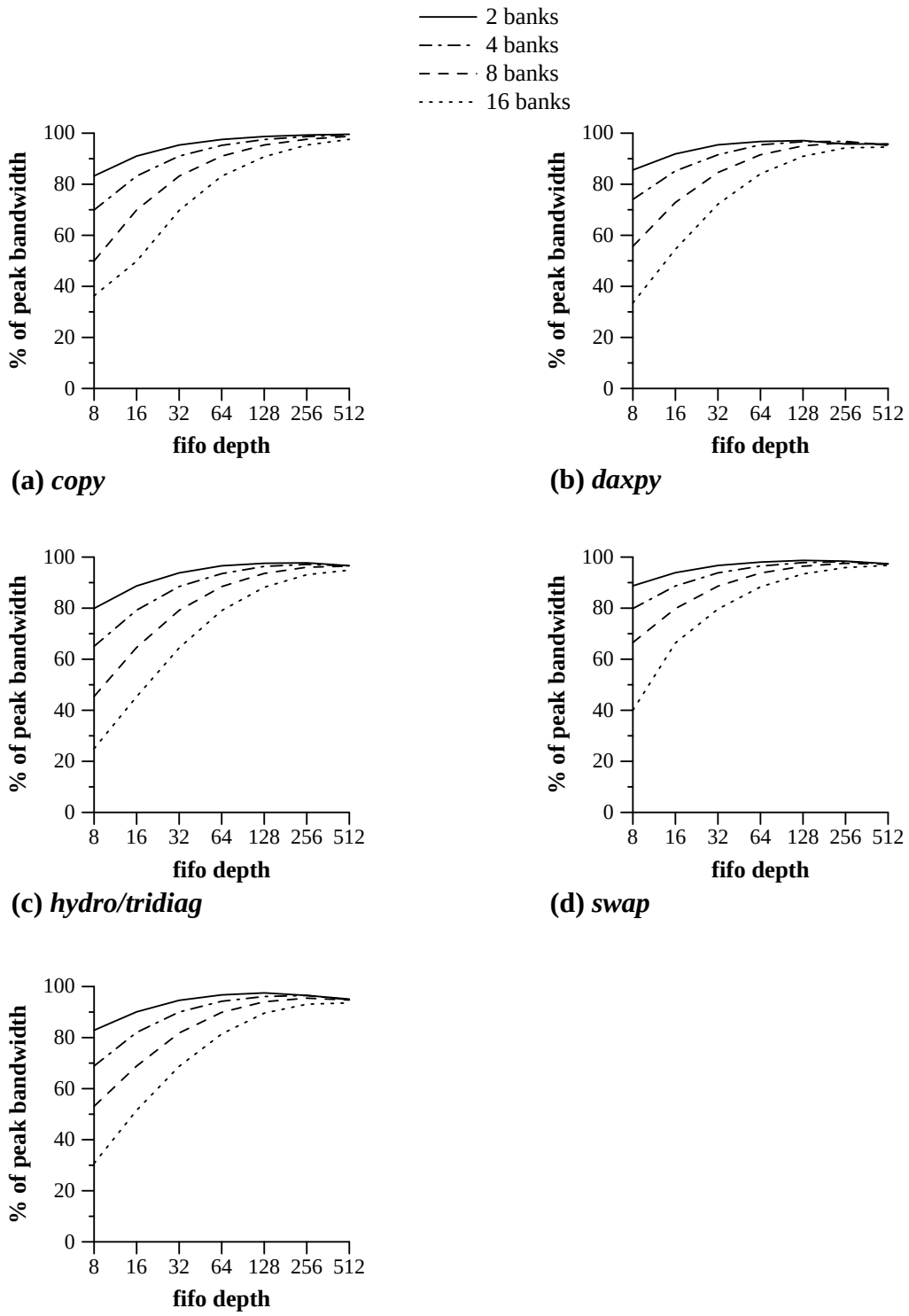



**Figure 86 Static Token TBC Performance for 8 CEs (Longer Vectors)**

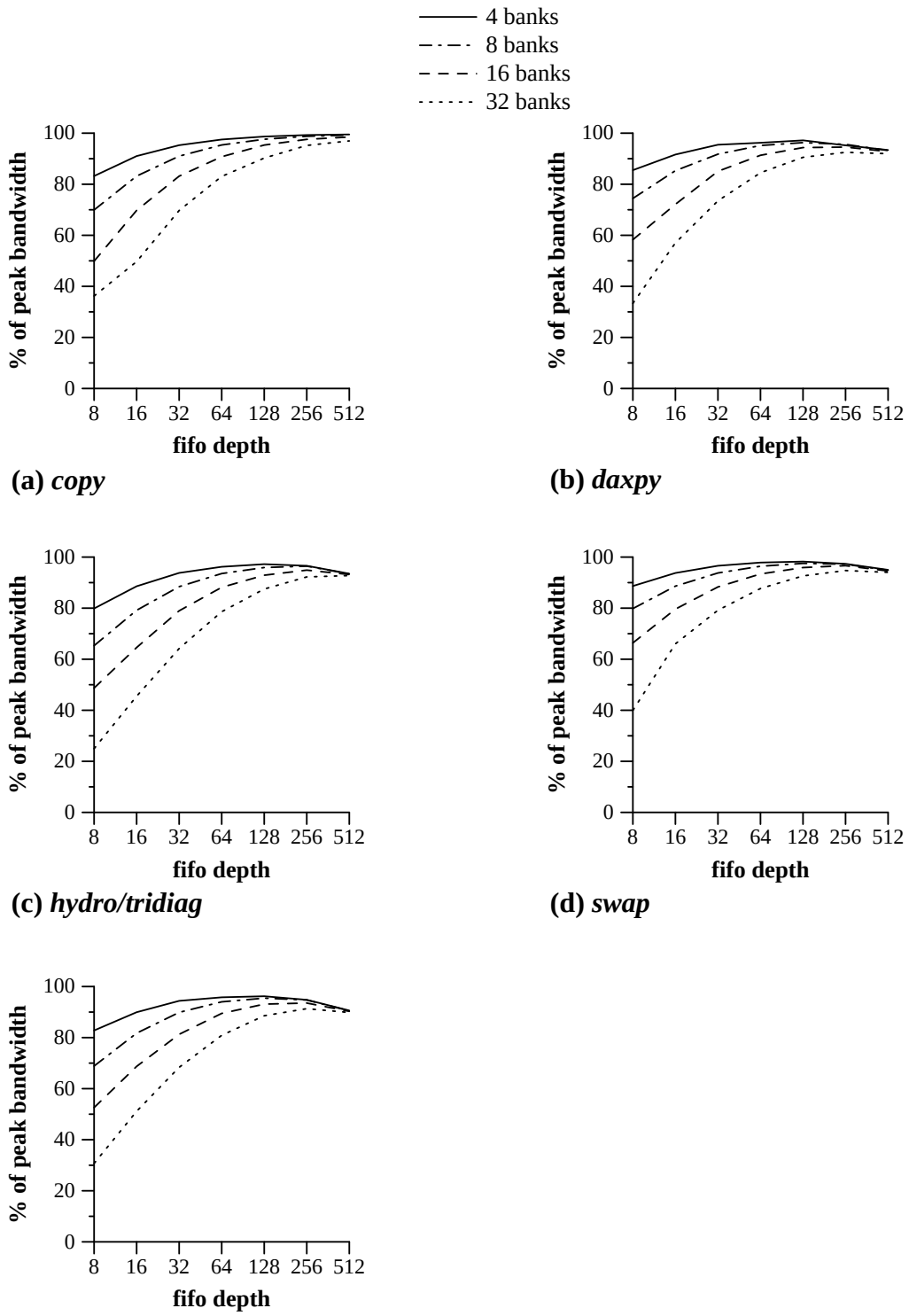



**(a)** ***copy*** **(b)** ***daxpy***

**(c)** ***hydro/tridiag*** **(d)** ***swap***

**(e)** ***vaxpy***

**Figure 87 Static Token TBC Performance for 2 CEs (Staggered Alignment)**




**(a)** ***copy***

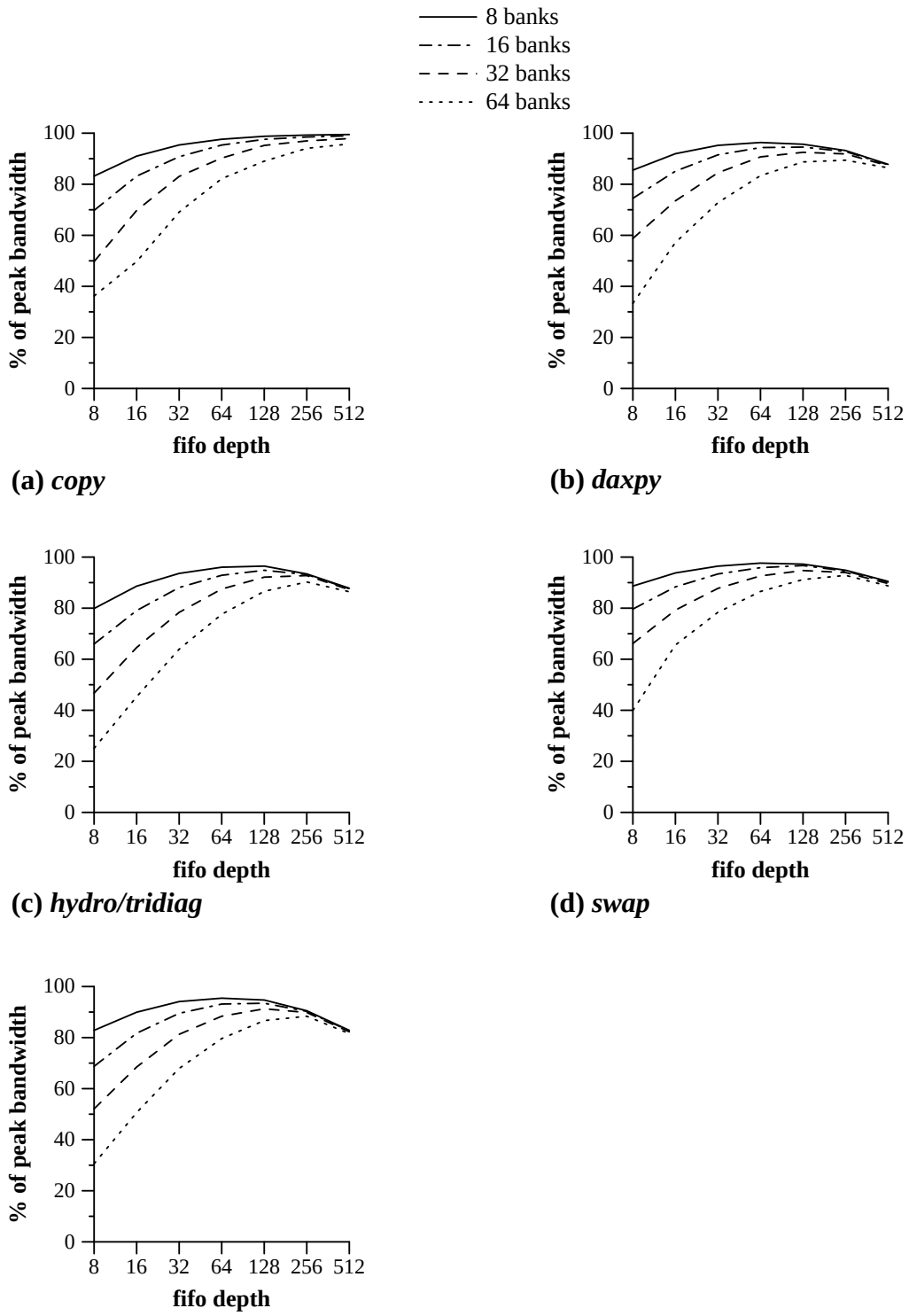
**(b)** ***daxpy***

**(c)** ***hydro/tridiag***

**(d)** ***swap***

**(e)** ***vaxpy***

**Figure 88 Static Token TBC Performance for 4 CEs (Staggered Alignment)**




**(a)** ***copy***

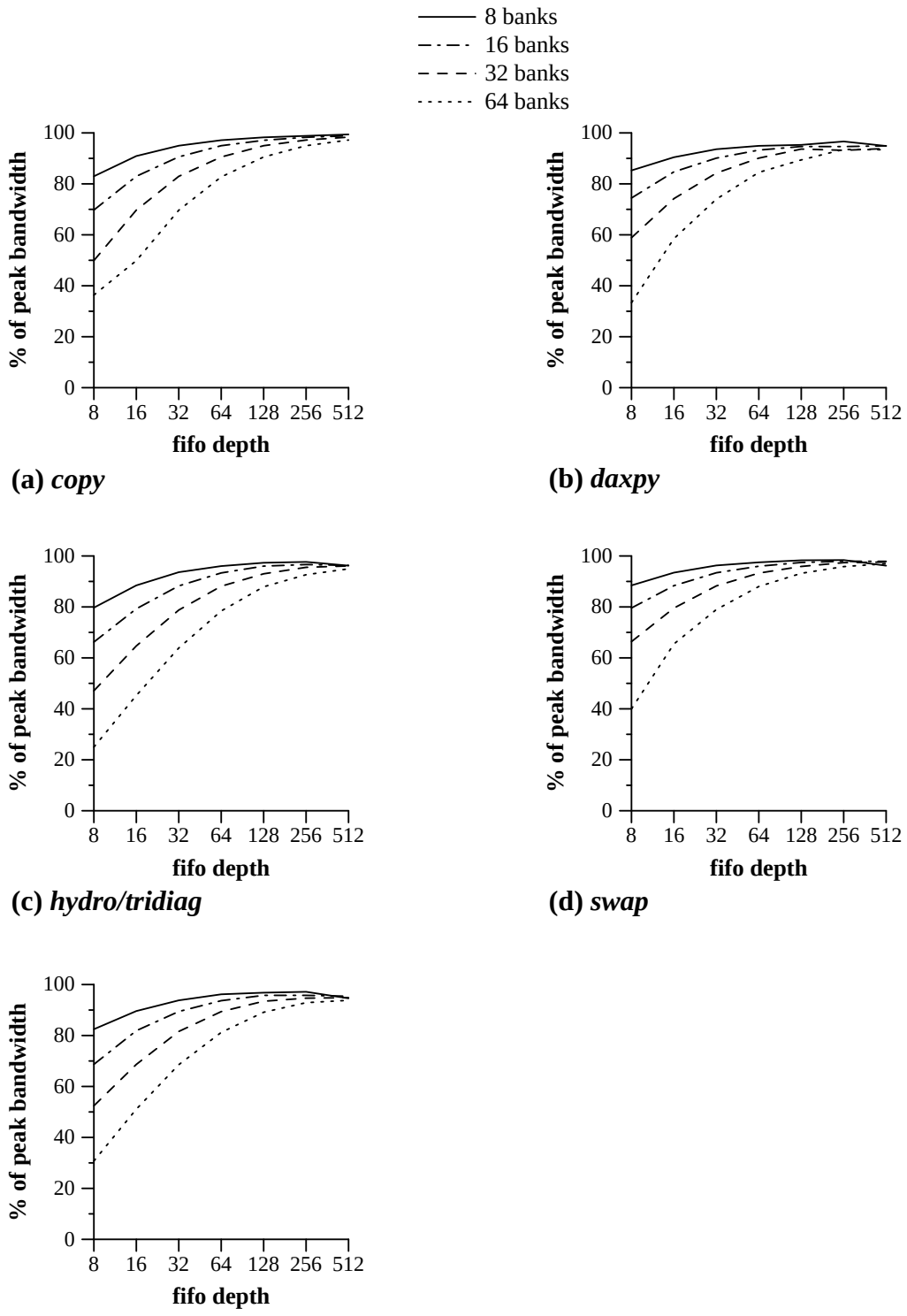
**(b)** ***daxpy***

**(c)** ***hydro/tridiag***

**(d)** ***swap***

**(e)** ***vaxpy***

**Figure 89 Static Token TBC Performance for 8 CEs (Staggered Alignment)**




**(a) *copy*** **(b) *daxpy***

**(c) *hydro/tridiag*** **(d) *swap***

**(e) *vaxpy***

**Figure 90 Static Token TBC Performance for 8 CEs (Longer Vectors, Staggered Alignment)**

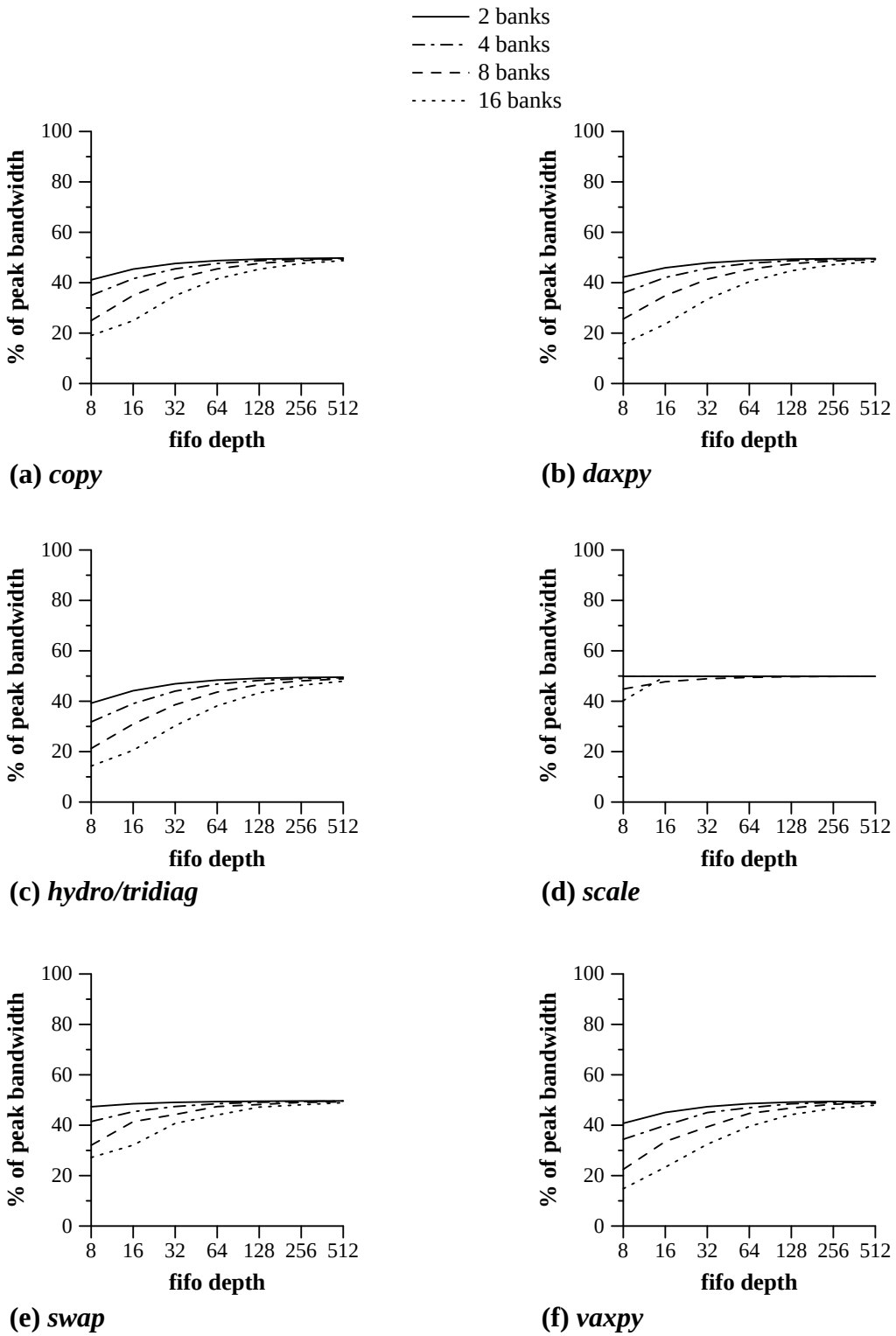



**Figure 91 Static FC Performance for 2 CEs**

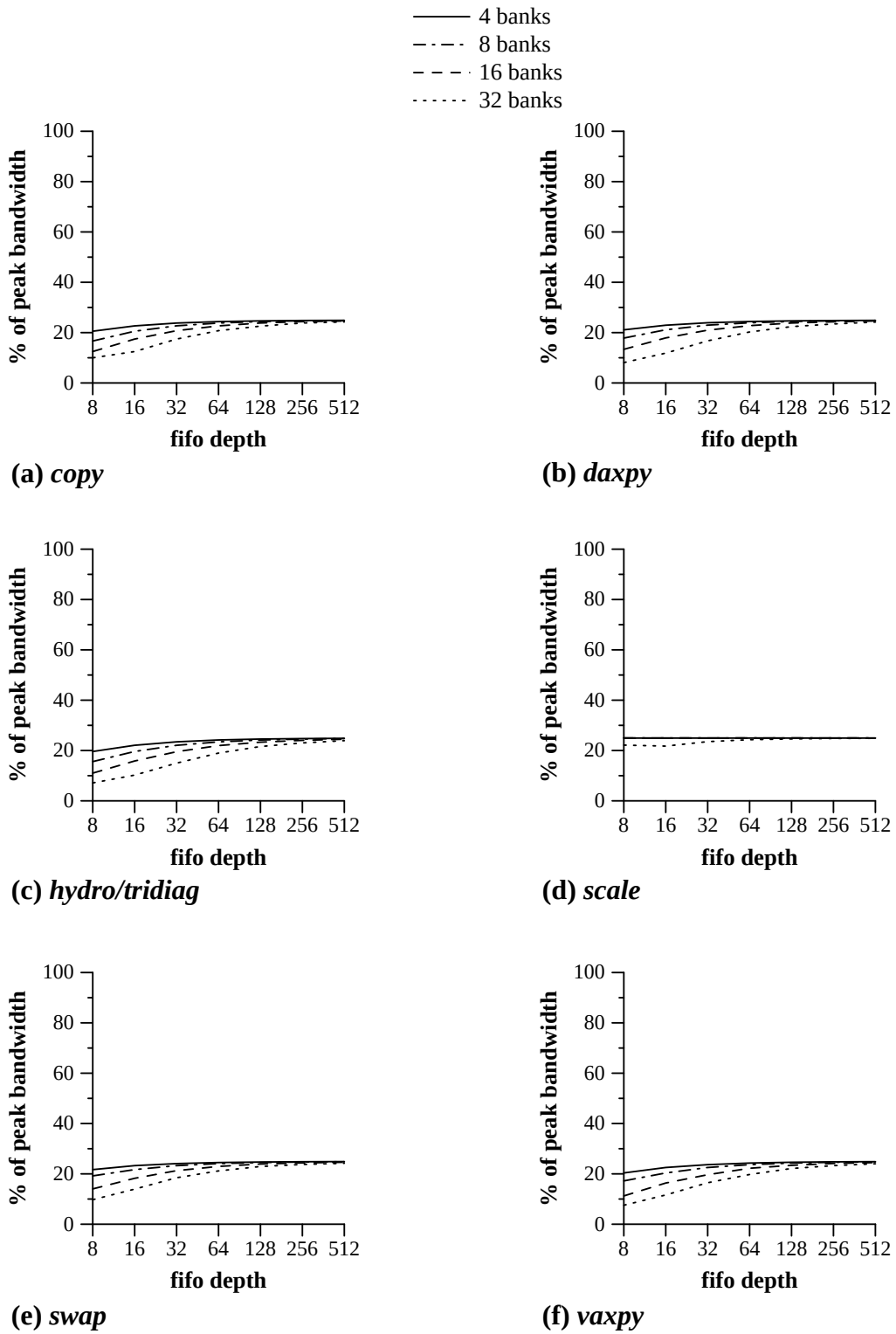



**Figure 92 Static FC Performance for 4 CEs**

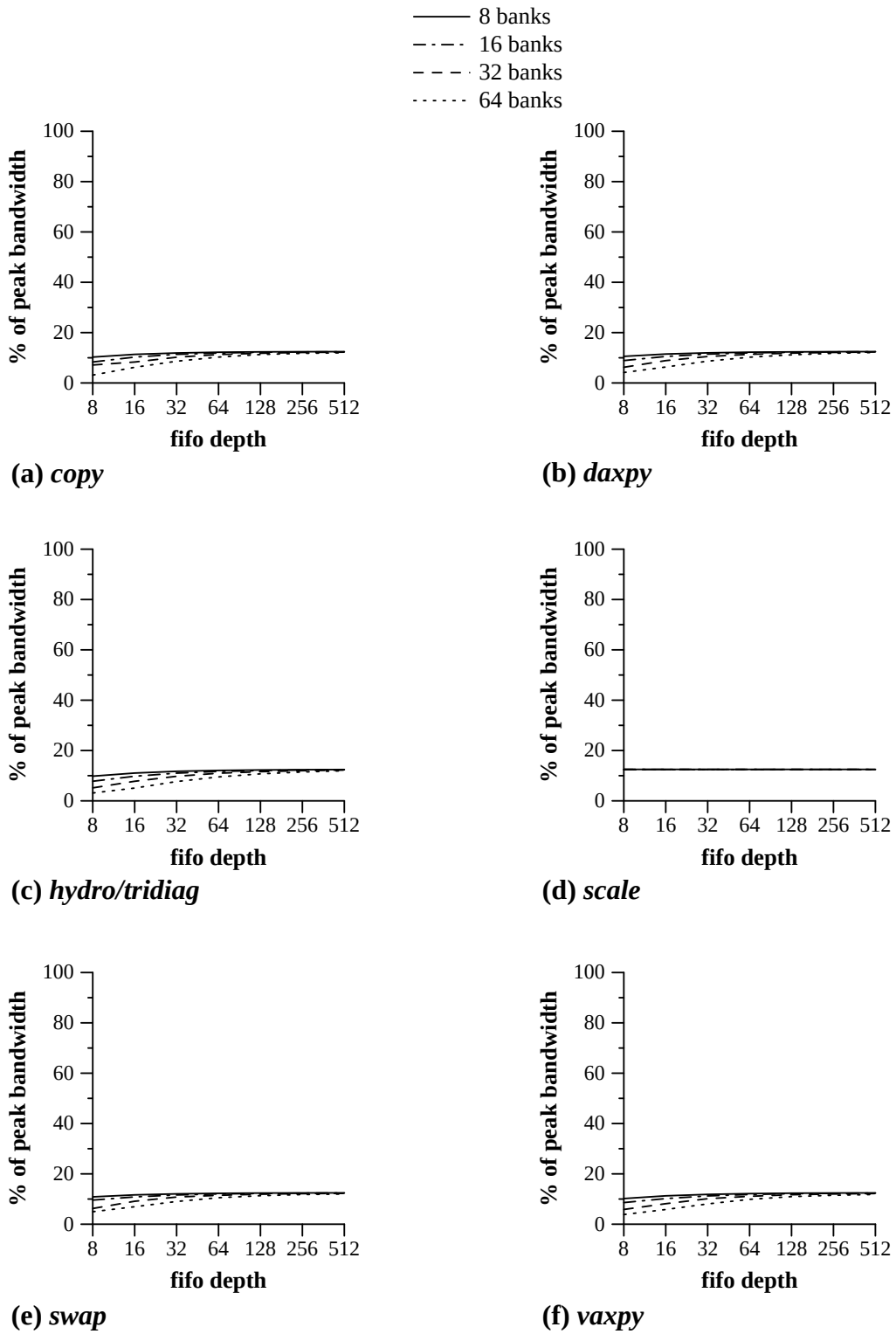



**Figure 93 Static FC Performance for 8 CEs**

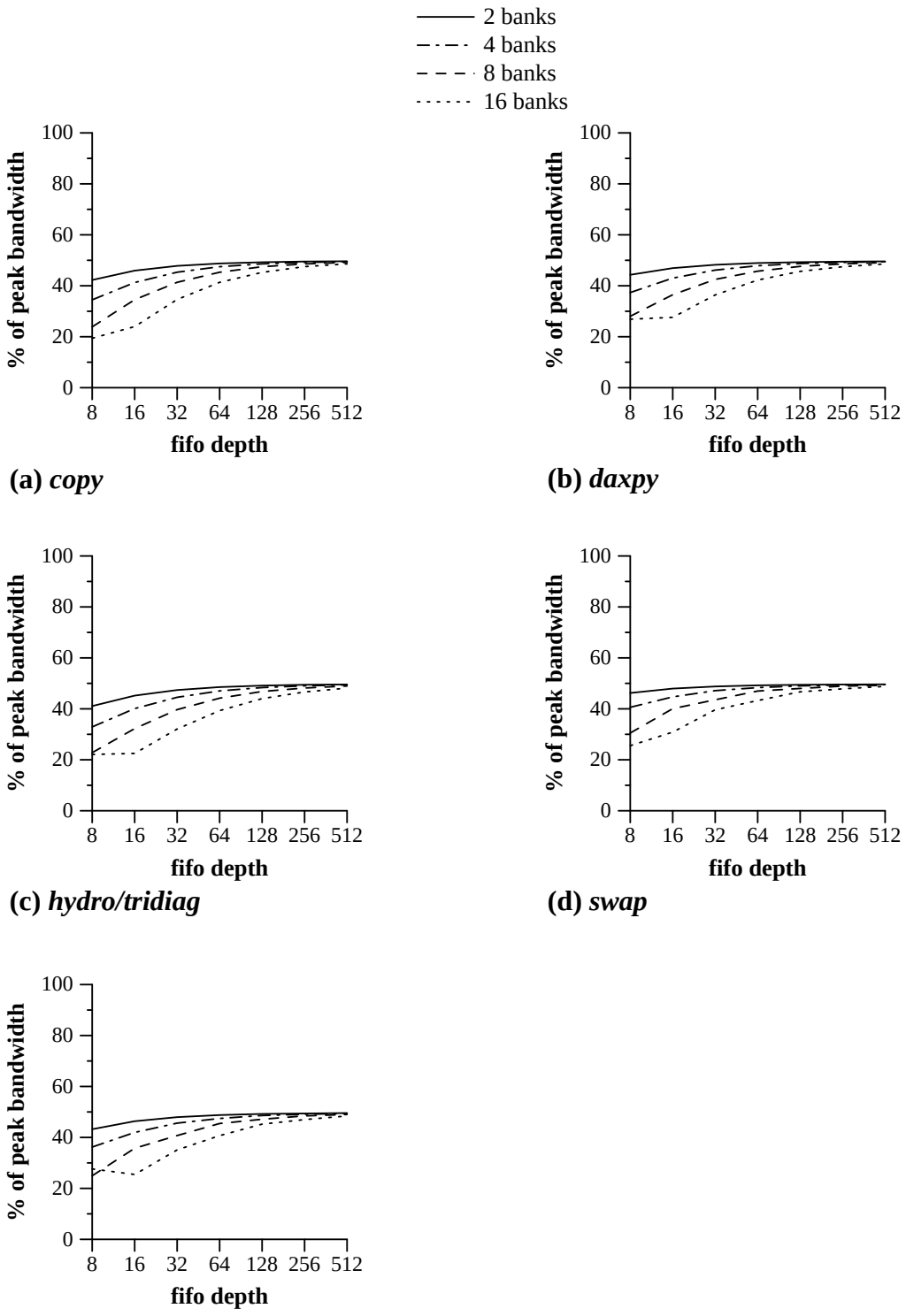



**Figure 94 Static FC Performance for 2 CEs (Staggered Alignment)**

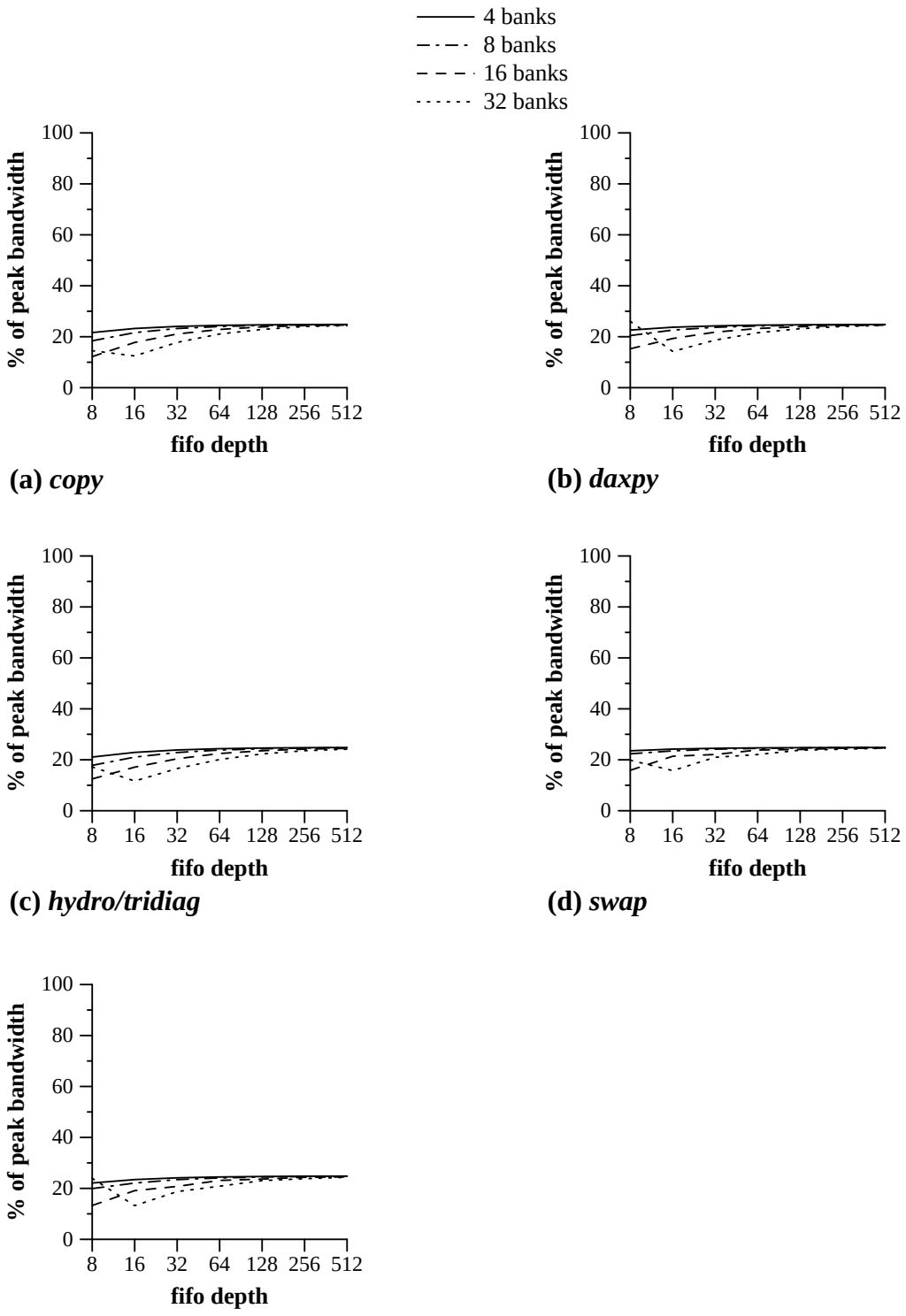
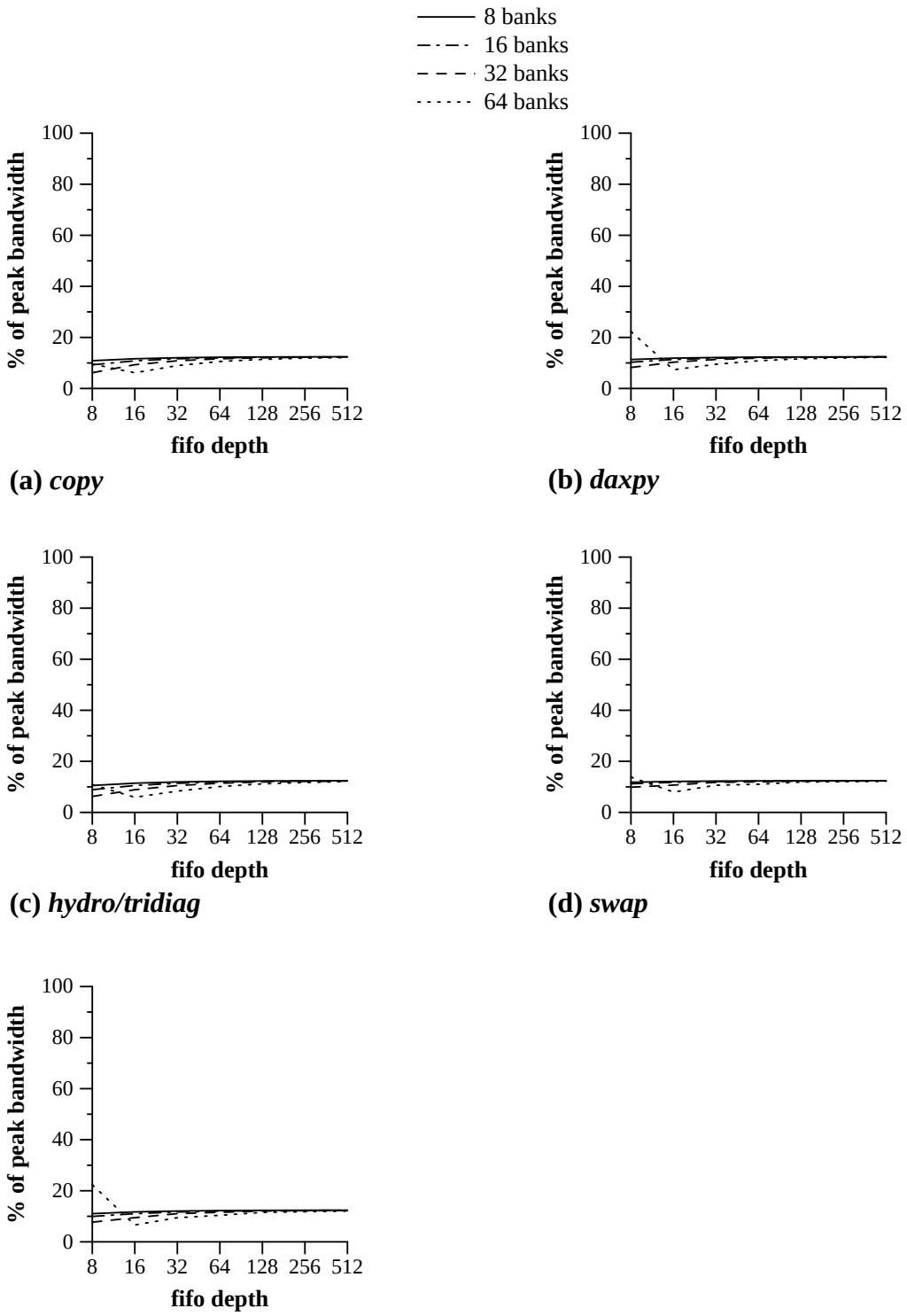



**Figure 95 Static FC Performance for 4 CEs (Staggered Alignment)**




**(a)** ***copy*** **(b)** ***daxpy***

**(c)** ***hydro/tridiag*** **(d)** ***swap***

**(e)** ***vaxpy***

**Figure 96 Static FC Performance for 8 CEs (Staggered Alignment)**

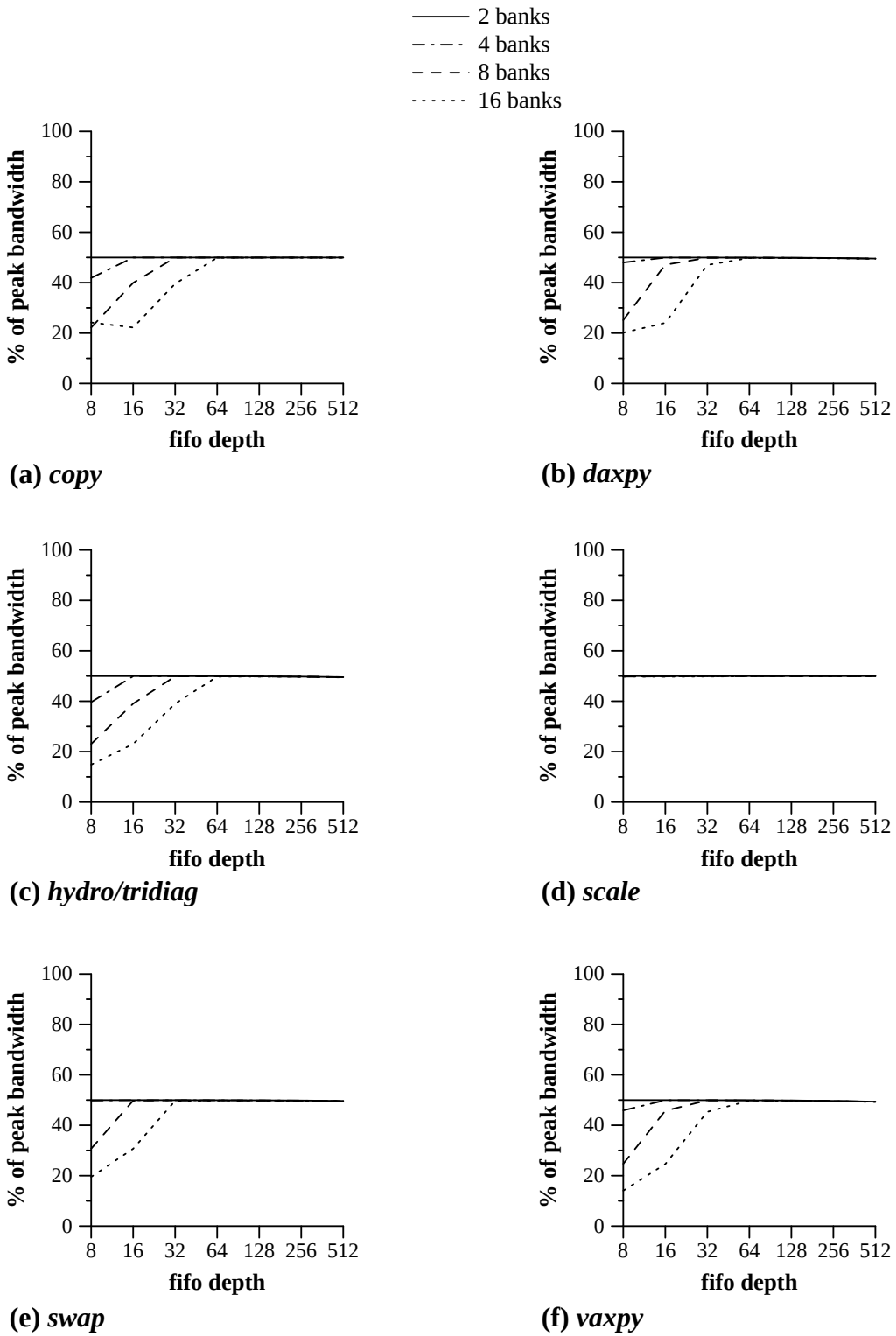
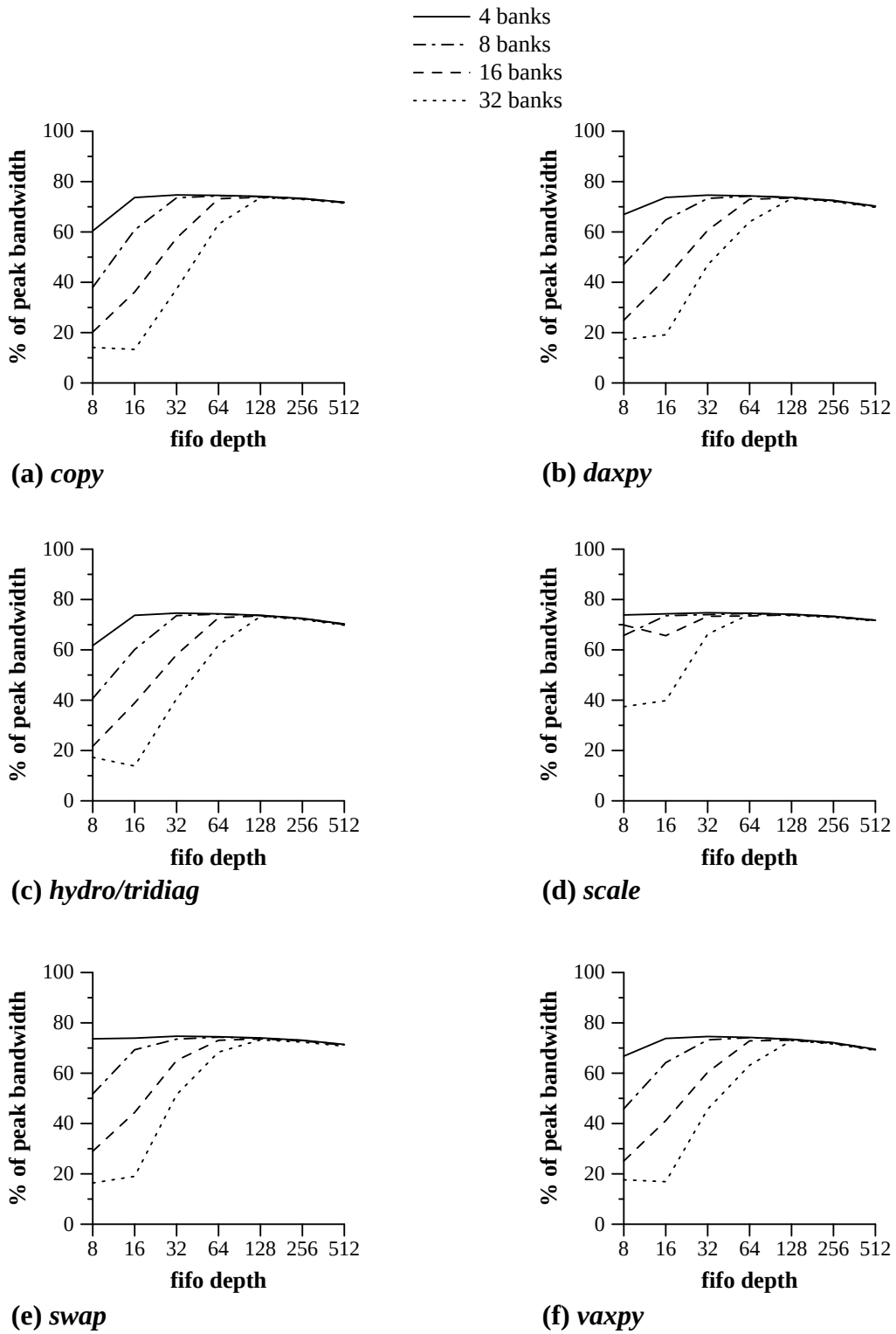



**Figure 97 Static FC Performance for 2-CE System Using 1 CE**




**(a)** ***copy*** **(b)** ***daxpy***

**(c)** ***hydro/tridiag*** **(d)** ***scale***

**(e)** ***swap*** **(f)** ***vaxpy***

**Figure 98 Static FC Performance for 4-CE System Using 3 CEs**

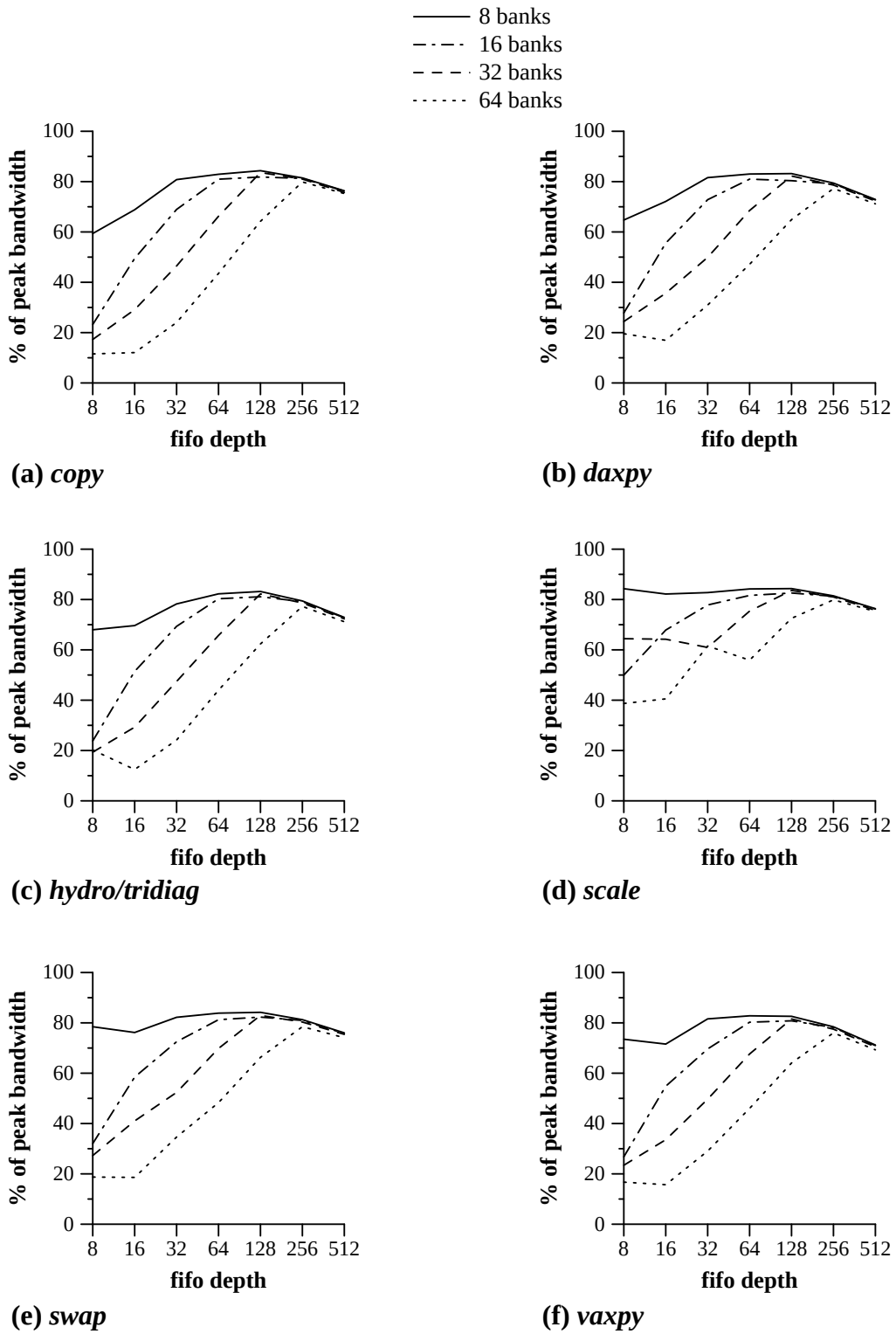
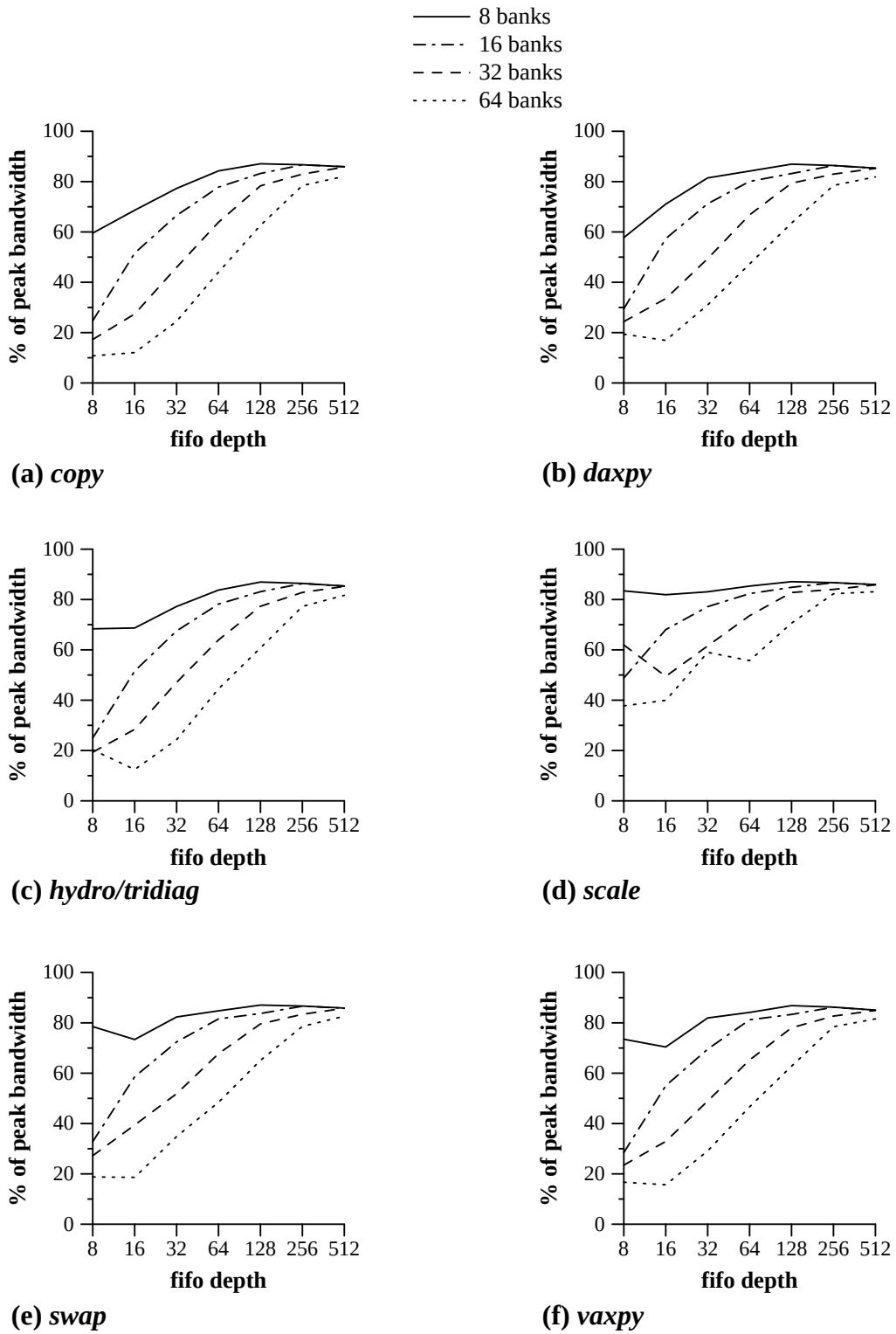



**Figure 99 Static FC Performance for 8-CE System Using 7 CEs**




**(a)** ***copy*** **(b)** ***daxpy***

**(c)** ***hydro/tridiag*** **(d)** ***scale***

**(e)** ***swap*** **(f)** ***vaxpy***

**Figure 100 Static FC Performance for 8-CE System Using 7 CEs (Longer Vec-**

## References

[Abu79] Abu-Sufah, W., Kuck, D.J., and Lawrie, D.H., “Automatic Program Transformations for Virtual Memory Computers”, Proc. 1979 National Computer Conference, June, 1979.

[Abu86] Abu-Sufah, W., and Malony, A., “Vector Processing on the Alliant FX/8 Multiprocessors”, Proc. 1986 International Conference on Parallel Processing, August, 1986.

[Bae91] Baer, J.L., and Chen, T.F., “An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty”, Proc. Supercomputing ‘91, November, 1991.

[Bud71] Budnik, P., and Kuck, D., “The Organization and Use of Parallel Memories”, IEEE Trans. Comput., 20, 12, 1971.

[Cal91] Callahan, D., Kennedy, K., and Porterfield, A., “Software Prefetching”, Proc. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, April, 1991.

[Car89] Carr, S., Kennedy, K., “Blocking Linear Algebra Codes for Memory Hierarchies”, Proc. Fourth SIAM Conference on Parallel Processing for Scientific Computing, 1989.

[Con90] “CONVEX Architecture Reference (C200 Series)”, CONVEX Computer Corporation Document No. 081-009220-000, 5th ed., September 1990.

[Dav91] Davidson, J.W., and Benitez, M.E., “Code Generation for Streaming: An Access/Execute Mechanism”, Proc. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, April, 1991.

[Dig92] *Alpha Architecture Handbook,* Digital Equipment Corporation, 1992.

[Don79] Dongarra, J.J., et. al., “Linpack User’s Guide“, SIAM, Philadelphia, 1979.

[Don90] Dongarra, J.J., DuCroz, J., Duff, I., and Hammerling, S., “A set of Level 3 Basic Linear Algebra Subprograms”, ACM Trans. Math. Softw., 16:1-17, 1990.

[Far92] Farmwald, M., and Morring, D., “A Fast Path to One Memory”, in [IEEE92], pp. 50-51, October 1992.

[Gal87] Gallivan, K., Jalby, W., Meier, U., and Sameh, A., “The Impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design”, Technical Report UIUCSRD 625, University of Illinois, 1987.

[Gan87] Gannon, D., and Jalby, W., “The Influence of Memory Hierarchy on

Algorithm Organization: Programming FFTs on a Vector Multiprocessor", in *The Characteristics of Parallel Algorithms*. MIT Press, 1987.

[Gao93] Gao, Q.S., "The Chinese Remainder Theorem and the Prime Memory System", Proc. 20th International Symposium on Computer Architecture, May 1993.

[Gol93] Golub, G., and Ortega, J.M., *Scientific Computation: An Introduction with Parallel Computing*, Academic Press, Inc., 1993.

[Gup91] Gupta, A., Hennessy, J., Gharachorloo, K., Mowry, T., and Weber, W.-D., "Comparative Evaluation of Latency Reducing and Tolerating Techniques", Proc. 18th International Symposium on Computer Architecture, May 1991.

[Har87] Harper, D. T., Jump, J., "Vector Access Performance in Parallel Memories Using a Skewed Storage Scheme", IEEE Trans. Comput., 36, 12, 1987.

[Har89] Harper, D. T., "Address Transformation to Increase Memory Performance", 1989 International Conference on Supercomputing.

[Har92] Hart, C., "Dynamic RAM as Secondary Cache", in [IEEE92], p. 48, October, 1992.

[Hen90] Hennessy, J., and Patterson, D., "Computer Architecture: A Quantitative Approach", Morgan Kaufmann, San Mateo, CA, 1990.

[IEEE92] "High-speed DRAMs", Special Report, IEEE Spectrum, vol. 29, no. 10, October 1992.

[Int91] *i860 XP Microprocessor Data Book*, Intel Corporation, 1991.

[Int92] *i860 Microprocessor Family Programmer's Reference Manual*, Intel Corporation, 1991.

[Jon92] Jones, F., "A New Era of Fast Dynamic RAMs", in [IEEE92], pp. 43-49, October, 1992.

[Kat89] Katz, R., and Hennessy, J., "High Performance Microprocessor Architectures", University of California, Berkeley, Report No. UCB/CSD 89/529, August, 1989.

[Lai92] Laird, M., "A Comparison of Three Current Superscalar Designs", Computer Architecture News, 20:3, June, 1992.

[Lam91] Lam, M., Rothberg, E., and Wolf, M., "The Cache Performance and Optimizations of Blocked Algorithms", Proc. ASPLOS-IV.

[Law79] Lawson, et. al., "Basic Linear Algebra Subprograms for Fortran Usage",

ACM Trans. Math. Soft., 5, 3, 1979.

[Lee91] Lee, K. “Achieving High Performance on the i860 Microprocessor”, NAS Technical Report RNR-91-029, NASA Ames Research Center, Moffett Field, CA, October 1991.

[Lee92] Lee, K. “On the Floating Point Performance of the i860 Microprocessor”, NAS Technical Report RNR-90-019, NASA Ames Research Center, Moffett Field, CA, July 1992.

[Lee93] Lee, K. “The NAS860 Library User’s Manual”, NAS Technical Report RND-93-003, NASA Ames Research Center, Moffett Field, CA, March 1993.

[Los92] Loshin, D., and Budge, D., “Breaking the Memory Bottleneck, Parts 1 & 2”, Supercomputing Review, January/February, 1992.

[Mea92] Meadows, L., Nakamoto, S., Schuster, V., “A Vectorizing Software Pipelining Compiler for LIW and Superscalar Architectures”, Proc. RISC 92.

[McK69] McKeller, A.C., and Coffman, E.G., “The Organization of Matrices and Matrix Operations in a Paged Multiprogramming Environment”, CACM, 12:3, 1969.

[McK93a] McKee, S.A, “Hardware Support for Access Ordering: Performance of Some Design Options”, University of Virginia, Department of Computer Science, Technical Report CS-93-08, August 1993.

[McK93b] McKee, S.A., “An Analytic Model of SMC Performance”, University of Virginia, TR CS-93-54, November, 1993.

[McK93c] McKee, S.A., “Uniprocessor SMC Performance on Vectors with Non-unit Strides”, University of Virginia, TR CS-93-67, December, 1993.

[McK94a] McKee, S.A., Klenke, R.H., Schwab, A.J., Wulf, Wm.A., Moyer, S.A., Hitchcock, C., Aylor, J.H., “Experimental Implementation of Dynamic Access Ordering”, Proc. HICSS-27, Maui, HI, January 1994; also University of Virginia, TR CS-93-42, August 1993.

[McK94b] McKee, S.A., Moyer, S.A., Wulf, Wm.A., Hitchcock, C., “Increasing Memory Bandwidth for Vector Computations”, Proc. Conf. on Prog. Lang. and Sys. Arch., Zurich, Switzerland, March 1994; also University of Virginia, TR CS-93-34.

[McM86] McMahon, F.H., “The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range”, Lawrence Livermore National Laboratory, UCRL-53745, December 1986.

[Mow92] Mowry, T.C., Lam, M., and Gupta, A., “Design and Evaluation of a Compiler Algorithm for Prefetching”, Proc. Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, September, 1992.

[Moy93] Moyer, S.A., “Access Ordering and Effective Memory Bandwidth”, Ph.D. Dissertation, Department of Computer Science, University of Virginia, Technical Report CS-93-18, April 1993.

[Ost89] Osterhaug, Anita, ed., *Guide to Parallel Programming on Sequent Computer Systems*, Prentice Hall, 1989.

[Ous90] Ousterhout, J. “Why aren’t Operating Systems Getter Faster As Fast As Hardware?” Proc. 1990 USENIX Summer Conference, June 1990.

[Por89] Porterfield, A.K., “Software Methods for Improvement of Cache Performance on Supercomputer Applications”, Ph.D. Thesis, Rice University, May, 1989.

[Qui91] Quinnell, R., “High-speed DRAMs”, EDN, May 23, 1991.

[Ram92] “Architectural Overview”, Rambus Inc., Mountain View, CA, 1992.

[Rau91] Rau, B. R., “Pseudo-Randomly Interleaved Memory”, 18th International Symposium on Computer Architecture, May 1991.

[Soh91] Sohi, G., and Franklin, M., “High Bandwidth Memory Systems for Superscalar Processors”, Proc. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, April, 1991.

[Tha81] Thabit, K.O., “Cache Management by the Compiler”, Ph.D. thesis, University of Illinois, October, 1982.

[Tem93] Temam, O., Granston, E.D., and Jalby, W., “To Copy of Not to Copy: A Compile-Time Technique for Assessing When Data Copying Shoud Be Used to Eliminate Cache Conflicts”, Proc. Supercomputing’93, December, 1993.

[Wal85] Wallach, S., “The CONVEX C-1 64-bit Supercomputer”, Compcon Spring 85, February 1985.

[Wol87] Wolfe, M., “Iteration Space Tiling for Memory Hierarchies”, Proc. Third SIAM Conference on Parallel Processing for Scientific Computing, December, 1987.

[Wol89] Wolfe, M., “More Iteration Space Tiling”, Proc. Supercomputing ‘89, 1989.