Dynamic Access Ordering: Bounds on Memory Bandwidth

Sally A. McKee

Computer Science Report No. CS-94-38
November 1, 1994

This work was supported in part by a grant from Intel Supercomputer Division and by NSF grants MIP-9114110 and MIP-9307626.

Dynamic Access Ordering: Bounds on Memory Bandwidth

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

Abstract

Memory bandwidth is becoming the limiting performance factor for many applications, particularly scientific computations. *Access ordering* is one technique that can help bridge the processor-memory performance gap. We are part of a team developing a combined hardware/software scheme for implementing access ordering dynamically at run-time. The hardware part of this solution is the *Stream Memory Controller*, or SMC. In order to validate the SMC concept, we have conducted numerous simulation experiments, the results of which are presented elsewhere. We have developed analytical models to bound asymptotic uniprocessor SMC performance, and have demonstrated that the simulation behavior of our dynamic access-ordering heuristics approaches those bounds. Here we introduce a model of of SMC startup costs, and we extend the uniprocessor SMC models to describe performance for modest-sized symmetric multiprocessor (SMP) SMC systems.

Dynamic Access Ordering: Bounds on Memory Bandwidth

1. Introduction

Memory speeds are increasing much more slowly than processor speeds [Kat89, Hen90]. As a result, memory bandwidth is rapidly becoming the limiting performance factor for many applications, particularly scientific computations. To illustrate the severity of the problem, consider using the 300 MHz, superscalar DEC Alpha to perform consecutive accesses to a DRAM with a 40ns cycle time: the processor can issue 24 or more instructions in the time it takes to complete a *single* memory access. A comprehensive, successful solution to the memory bandwidth problem must exploit the richness of the *full* memory hierarchy. Not only must we find ways to improve cache performance, but we must provide alternatives for computations for which caching is insufficient.

For instance, most memory devices manufactured in the last decade provide special capabilities that make it possible to perform some access sequences faster than others [IEEE92, Ram92, Qui91], and exploiting these component characteristics can dramatically improve effective bandwidth. For applications that perform vector-like memory accesses, bandwidth can be increased by reordering the requests to take advantage of device properties such as fast-page mode. Page-mode devices behave as if implemented with a single on-chip cache line, or *page*. A memory access that falls outside the address range of the current page forces a new page to be set up. The overhead time required to do this makes such *far* accesses significantly slower than *near* accesses that hit the current page.

Access ordering is any technique that changes the order of memory requests to increase bandwidth, but here we are specifically concerned with ordering vector-like *stream* accesses to exploit page-mode DRAMs. McKee, et. al., propose a combined hardware/ software scheme for implementing access ordering dynamically at run-time, and present

numerous simulation results demonstrating its effectiveness [McK94a]. The hardware part of this solution is the *Stream Memory Controller* (SMC). An analytical model to bound asymptotic SMC performance for unit-stride vectors has been developed and extended for non-unit stride vectors in [McK93b, McK93c]. Here we develop a model to bound SMC performance on short vectors, and we extend the asymptotic model to describe symmetric multiprocessor (SMP) SMC performance.

Note that we shall use the terms *vector* and *stream* interchangeably when doing so causes no confusion: a read-vector is equivalent to a read-stream, but a vector that is read, modified, and written constitutes two streams, a read-stream and a write-stream. The rest of this report is organized as follows. Section 2 presents the basic SMC architectures for uniprocessor and shared-memory multiprocessor systems. Section 3 provides an overview of the task-scheduling strategies we used to parallelize workloads for our SMP systems. Section 4 discusses the assumptions underlying both the startup delay model presented in Section 5 and the asymptotic performance models of Section 6. Section 7 and Section 8 discuss the environment and benchmark kernels used in the simulation studies of SMC performance [McK93a, McK93c, McK94c], and Section 9 correlates the performance curves generated by our analytic models with sample simulation results.

2. The SMC

Moyer develops algorithms and analyzes the performance benefits and limitations of doing compile-time access ordering [Moy93]. His scheme involves unrolling loops and grouping accesses to each stream, so that the cost of each DRAM page-miss can be amortized over several accesses that hit the current page. The extent to which this technique can be applied is limited by the size of the processor's register file, and an optimal ordering cannot be generated without the address alignment information usually available only at run time. These limitations motivate us to consider an implementation that reorders accesses dynamically. Benitez and Davidson's algorithm can be used to detect streams at compile-

time [Ben91], and the stream parameter can be transmitted to the reordering hardware at run time. What follows is an overview of the dynamic access ordering architecture proposed in the uniprocessor and SMP SMC reports [McK93a, McK93c, McK94c].

Our discussion is based on the simplified architectures of Figure 1 and Figure 2. In these systems, memory is interfaced to the CPU or *computational elements* (CEs) 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.

The MSU has full knowledge of all streams currently needed by the CEs: 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.

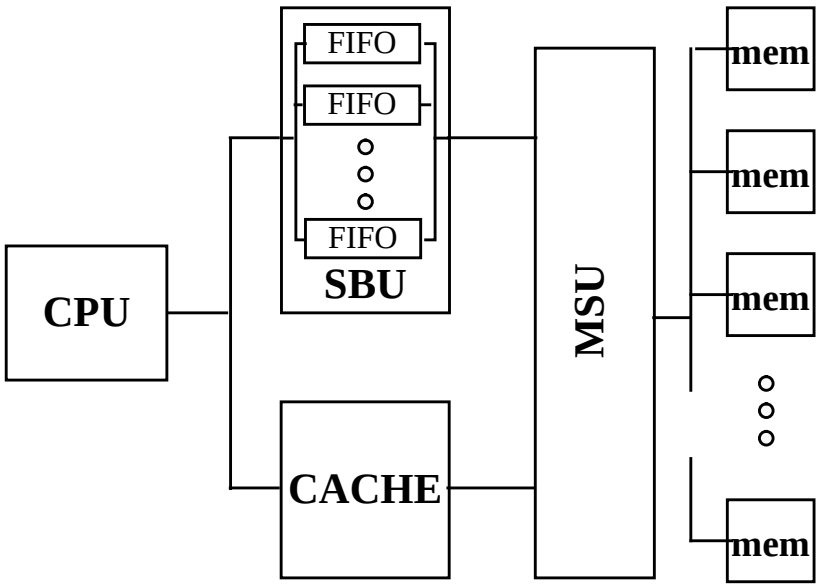


Figure 1 Uniprocessor SMC Organization

In the uniprocessor organization of Figure 1, a separate Stream Buffer Unit (SBU) contains high-speed buffers for stream operands and provides memory-mapped control registers that the processor uses to specify stream parameters (base address, stride, length, and data size). Together, the MSU and SBU comprise a Stream Memory Controller (SMC) system.

The stream buffers are implemented logically as a set of FIFOs within the SBU, as illustrated in Figure 1. Each stream is assigned to one FIFO, which is asynchronously filled from or drained to memory by the access/issue logic of the MSU. The “head” of the FIFO is another memory-mapped register, and load instructions from or store instructions to a particular stream reference the FIFO head via this register, dequeueing or enqueueing data as is appropriate.

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. Multiple-chip designs require inter-chip buses for communication. Such buses tend to be expensive, and the cost of moving data off-chip may prove to be the limiting performance factor with respect to the SMC’s operating speed. Since we seek to formulate bounds on SMP SMC performance, and since the most efficient hardware organization is one in which the entire SMC system and all computational elements reside on a single chip, that is the organization we consider here.

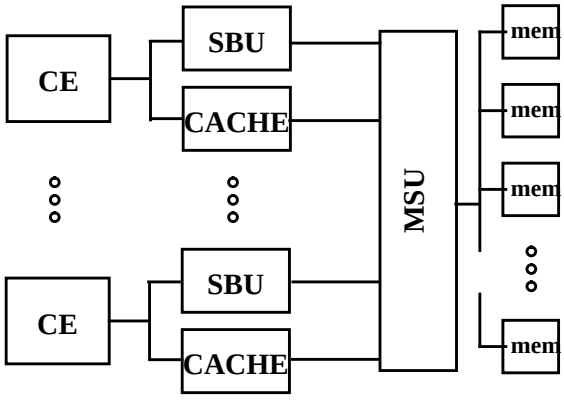


Figure 2 Symmetric Multiprocessor SMC Organization

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.

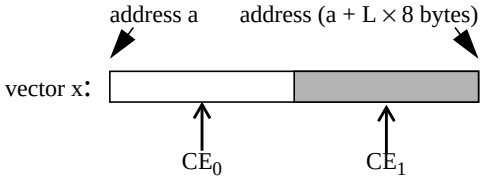
3. Task Scheduling

The way in which a problem is partitioned for a multiprocessor system can have a marked effect on performance. In particular, 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 near accesses can be issued to it in succession. Note that distinct vectors in the computation are assumed to have no DRAM pages in common.

Note that the number of memory banks in the system affects when DRAM page misses occur in a computation. To see this, consider a DRAM component in which each page holds 512 double-word vector elements. Then 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.

Three general scheduling techniques are commonly used to parallelize workloads: *prescheduling*, *static scheduling*, and *dynamic scheduling* [Ost89]. 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. Each vector is split into approximately equal-size chunks, and each CE processes a chunk. Figure 3 depicts this data distribution for a stride-one vector, along with 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 3 Prescheduling: Data Distribution for 2-CE System

On a 2-CE system with 8 banks, prescheduling divides a 10,000-element vector so that each CE processes approximately 5000 elements, as pictured in Figure 4. The vector chunks for each CE have been arranged vertically to emphasize the portions of data that are being processed in parallel. CE_1 crosses from DRAM page B to DRAM page C after 3192 loop iterations, but CE_0 doesn't begin using DRAM page B until it has completed 4096 iterations. Unless CE_1 proceeds much more slowly than CE_0, it is unlikely that the two computational elements will ever share pages during the computation

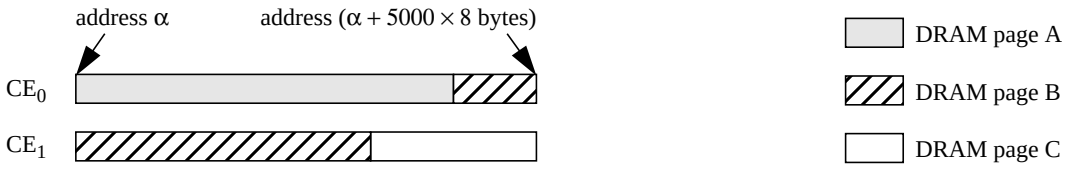


Figure 4 Distribution of 10,000-Element Vector for 8 Banks and 2 CEs

Figure 5 shows the distribution of the same 10,000-element vector on a 4-CE system with 8 banks. Now CE_0 and CE_1 use the same DRAM pages for almost two-thirds of the computation, and CE_3 and CE_4 share for the initial one-third. At the end, CE_2 and CE_3 will be on the same pages.

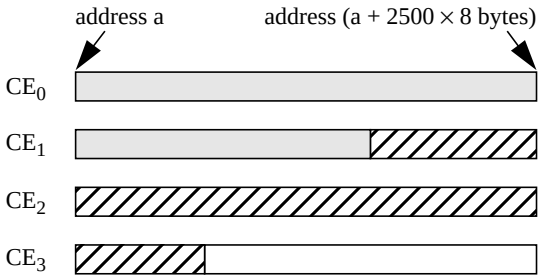
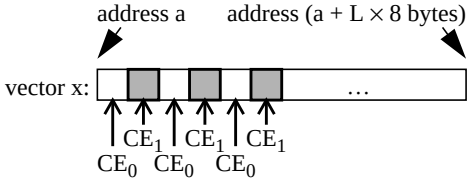


Figure 5 Distribution of 10,000-Element Vector for 8 Banks and 4 CEs

In static scheduling, tasks are divided among the CEs at run-time, but the partitioning is performed in some predetermined way. A process on a CE determines which tasks it must do, performs that work, then waits for other processes to finish their tasks. We model static scheduling by distributing loop iterations among the CEs, as in a FORTRAN DOALL loop. This parallelization scheme makes the *effective stride* at each of the M participating CEs M times the original stride of the computation. If the number of memory banks is a multiple of the number of CEs, this means that a different subset of banks will provide all the data for each CE. Figure 6 illustrates the data distribution and code for this scheme. With this model of static scheduling, each of M CEs performs every Mth iteration, thus all CEs use the same DRAM pages throughout most of the computation (if the CEs proceed at different rates, then some may cross page boundaries sooner than others).

Note that a static scheduling scheme could break the vector data into chunks instead of interleaving iterations across the CEs; SMC performance for such a scheme is identical to that for the prescheduling scheme described above.



CE0's code:
```
for (i = 0; i < L; i += 2) {
   /* operations on x[i]*/
}
```

CE1's code:
```
for (i = 1; i < L; i += 2) {
   /* operations on x[i]*/
}
```

Figure 6 Static Scheduling: Data Distribution for 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. Since we are only concerned with 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 a separate discussion of performance under dynamic scheduling.

4. Modeling Assumptions

Given an SMC system whose memory is composed of interleaved banks of page-mode DRAMs, we want to determine the peak achievable memory performance. The complex interactions between the many parameters of the SMC and memory system make it difficult to formulate a provably optimal dynamic ordering algorithm. Moreover, implementing such an algorithm might be expensive, both in the amount of hardware necessary and in the time required for it to run. Instead, we have developed a number of heuristics for dynamic access ordering; simulation results for these are presented elsewhere [McK93a, McK93c, McK94c].

Although we do not know precisely what the optimal ordering algorithm is, we *can* bound its performance. Taking advantage of the full bandwidth afforded by the memory system requires exploiting the page-mode capabilities of the memory components. Since bandwidth is limited by the number of page-misses incurred during a computation, we can derive a bound on SMC performance by calculating the minimum number of page-misses for that computation. We can then use this bound to evaluate the performance of our heuristics. Similarly, we can calculate the minimum time for a CE to execute a loop by adding the minimum time the CE must wait to receive all the operands for the first iteration to the minimum time required to execute all remaining instructions.

These two calculations provide us with two different bounds on SMC performance; the first illustrates asymptotic performance limits for very long vectors, and the second describes performance limits due to startup effects. In effect, the asymptotic model bounds bandwidth utilization between the SMC and memory, whereas the startup-delay model bounds bandwidth utilization between the CEs and the SMC.

As a practical consideration, we assume that the system is matched so that bandwith between the CEs and SMC does not exceed bandwidth between the SMC and memory. The vectors we consider are of equal length and share no DRAM pages in common, and we assume a model of operation in which each CE accesses its FIFOs in round-robin order, consuming one data item from each FIFO in each iteration.

In order that the bound we derive be conservative, we impose the following constraints. We ignore bus turnaround delays and other external effects. We model the CE as a generator of non-cached loads and stores of vector elements; all other computation is assumed to be infinitely fast, putting as much stress as possible on the memory system. In calculating the number of page misses incurred by a multiple-stream computation, we assume that DRAM pages are infinitely large. In other words, we assume that misses resulting from crossing page boundaries are subsumed by the other misses calculated in our model. Finally, we derive our performance bounds by assuming that the SMC always amortizes page miss costs over as many accesses as possible: read FIFOs are completely empty and write FIFOs are completely full whenever the SMC begins servicing them.

We first look at how SMC startup costs impact overall performance, then we examine the limits of the SMC's ability to amortize page-miss costs as vector length increases asymptotically. We develop each of these models for uniprocessor SMC systems, then extend them to describe multiprocessor SMC performance.

5. The Startup-Delay Model

Whereas the traditional performance concern has been to optimize *processor* utilization, here we focus on computations and systems that are primarily bandwidth-limited, thus we strive to optimize *bandwidth* utilization. Nonetheless, good overall performance requires that the computational element(s) not be left unnecessarily idle. For instance, the rate at which a CE removes data from a read FIFO affects the amount of buffer space available, which in turn limits the number of near accesses over which the MSU can amortize page-miss costs.

Recall that the bandwith between the CE and SMC equals that between the SMC and memory, thus optimal system performance allows each CE to complete one memory access each bus cycle.

Since the Memory Scheduling Unit (MSU) attempts to issue as many accesses as possible to the current DRAM pages, most of our dynamic access-ordering heuristics tend to fill the currently selected FIFO(s) completely before moving on to service others. At the beginning of a computation, this means that a CE stalls waiting for the first element of the *n*th vector while the MSU fills the FIFOs for the first $n - 1$ vectors. By the time the MSU has provided all the operands for the first loop iteration, it will also have prefetched enough data for many future iterations, thus the computation can proceed without stalling the CE again soon. Deeper FIFOs cause the CE to wait longer at startup, but if the vectors in the computation are sufficiently long, the deep FIFOs allow these startup delays to be amortized over enough accesses to make them insignificant. Unfortunately, for shorter vector computations there are many fewer accesses over which to amortize startup costs. In such cases, the initial delays represent a significant portion of the total time for the computation.

Consider an SMC with FIFOs of depth *f*. For a computation involving two read vectors of length $l = f$, the CE must wait *f* cycles (while the first FIFO is being filled) between

reading the first operand of the first vector and the first operand of the second vector. According to our model (in which arithmetic and control are assumed to be infinitely fast), the actual processing of the data requires $2f$ cycles, one cycle to read each element in each vector. Thus, without even considering DRAM page misses, the total time for a computation is the time to fetch the first iteration's operands plus the time to finish processing all data. For this particular system and computation, the time is at best $f + 2l = f + 2f = 3f$ cycles. This is only 66% of the optimal performance of $2l = 2f$ cycles (i.e., the time to process $2l$ vector elements). Figure 7 presents a timeline of the computation: the processor and memory both require the same number of cycles to do their work, but the extent to which their activities overlap determines the time to completion.

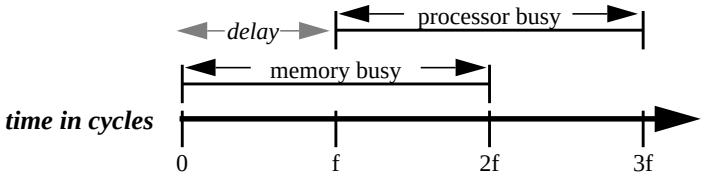


Figure 7 Startup Delay for a Computation with 2 Read Streams of Length *l* = *f*

Let s represent the number of streams in a computation, and let s_{read} represent the number of read streams. The bandwidth limits caused by startup delays can then be described by:

$$\%\ \text{peak bandwidth}\ \frac{s \times l}{f(s_{read} - 1) + (s \times l)} \times 100.0 = \frac{s}{\left(\frac{f}{l}\right)(s_{read} - 1) + s} \times 100.0 \qquad (1)$$

Figure 8 illustrates these limits as a function of the log of the ratio of fifo depth to vector length for a uniprocessor reading two streams and writing one.

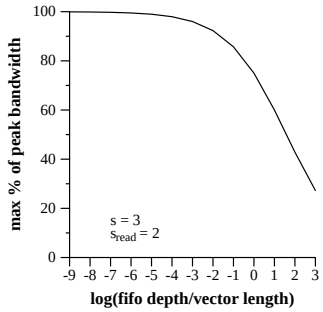


Figure 8 Performance Limits Due to Startup Delays for 1 Write and 2 Read Streams

In an SMP environment, we can bound the performance of the entire parallel computation by first calculating the startup delay for the last computational element to begin its share of the processing, and then adding the minimum time for that CE to execute its remaining iterations. In developing these formulas, we assume that all CEs are performing the same operation, but are acting on different data. The vector length l reflects the portion of each vector being processed by a single CE.

We can derive tighter bounds by tailoring our model to a particular SMC implementation. The way in which the MSU fills the FIFOs affects how long the CEs must wait to receive the operands for their first iteration. If the MSU's ordering heuristic only services one FIFO at a time, then the last CE must wait while the MSU fetches the read streams for all other CEs and all but one of its own read streams. On the other hand, if the MSU can service more than one FIFO at a time, more than one CE can start computing right away.

In the former case, the minimum number of cycles required to fill a FIFO is $1/N$ times the minimum for a uniprocessor system (because the bandwidth of the system is balanced, and there are now N CEs that can each execute a memory reference per cycle). Let M represent the number of CEs participating in the computation. Then the CEs are using M/N times the potential bandwidth, and the number of streams that must be fetched before the last CE can start is $(M \times s_{read}) - 1$. The startup-delay formula under these circumstances is:

$$\frac{s}{\left(\frac{1}{N}\right)\left(\frac{f}{l}\right)(Ms_{read} - 1) + s} \times \frac{M}{N} \times 100.0 \qquad (2)$$

For the latter case, let us assume that the MSU can perform accesses to M FIFOs at a time (one FIFO for each participating CE). When $M = N$, the formula for startup delays is the same as for the uniprocessor SMC system (Equation 1). To see this, note that each CE need only wait for all but one of its own read streams to be fetched, and the average rate at which those FIFOs are filled will be one element per processor cycle. When $M < N$, the average time to fill a FIFO will be M/N times that for a uniprocessor, and the formula becomes:

$$\frac{s}{\left(\frac{M}{N}\right)\left(\frac{f}{l}\right)(s_{read}-1)\ +s}\times\frac{M}{N}\times 100.0\ =\ \frac{s}{\left(\frac{1}{N}\right)\left(\frac{f}{l}\right)(Ms_{read}-M)\ +s}\times\frac{M}{N}\times 100.0 \quad (3)$$

Thus the startup delays for the two cases only differ by a factor of $M-1$. Note that the Equation 3 also represents a bound on bandwidth for the case where the MSU fills a single FIFO at a time, thus we will use it as the basis for comparison with our simulation results.

6. Asymptotic Models

If a computation's vectors are long enough to make SMC startup costs a negligible portion of the memory time for the computation, then the limiting performance factor becomes the number of near (fast) accesses we can make. The following models calculate the minimum number of DRAM page misses that a computation must incur.

6.1 Uniprocessor Models

We will refer to a read-only or write-only vector as a *single-access vector.* Likewise, a read-modify-write vector will be referred to as a *double-access vector.* The terms *stream* and *FIFO* will be used interchangeably, since we have assumed an SMC model in which each stream is assigned to exactly one FIFO. In the following, let "FIFO" refer to a read FIFO unless otherwise stated; the analysis for write FIFOs is analogous. We first present a model for small-stride, multiple-vector computations; we then extend this model to describe single-vector computations and multiple-vector computations with large strides.

Multiple-Vector Computations

Let b be the number of interleaved memory banks, and let f be the depth of the FIFOs. Every time the MSU switches FIFOs, it incurs a page miss in *each* memory bank, thus the percentage of accesses that cause DRAM page misses for a stream whose stride is relatively prime to the number of banks is at least b/f. Strides not relatively prime to the number of banks prevent us from exploiting the full system bandwidth, for such vectors don't hit all banks. In calculating performance for these vectors, we must adjust our formulas to reflect

the percentage of banks actually used. The number of banks used is $b/gcd(b, stride)$, thus the fraction of accesses that miss the page is at least $\frac{b}{gcd(b, stride) \times f}$.

Let v be the number of distinct vectors in the computation, and let s be the number of streams. If the CE accesses the FIFOs (in round robin order) at the same rate as the memory system, then while the MSU is filling a FIFO of depth f, the CE will consume f/s more data elements from that stream, freeing space in the FIFO. While the MSU supplies f/s more elements, the CE can remove f/s^2, and so on. Our equation for calculating the miss rate for single-access vectors becomes:

$$r = \frac{b}{gcd(b, stride)} \times \frac{1}{f\left(1 + 1/s + 1/s^2 + 1/s^3 + \ldots\right)} \tag{4}$$

In the limit, the series in the denominator of the second term converges to $s/(s-1)$, and our equation reduces to $r = \frac{b(s-1)}{gcd(b, stride) \times fs}$.

The number of page misses for a double-access vector is the same as for a single-access vector, but the read-modify-write vector is accessed twice as many times and requires two FIFOs, one for the read stream and one for the write stream. Thus for these vectors, the *percentage* of accesses that cause page misses is *half* that of a single-access vector.

To calculate the average DRAM page-miss rate for the entire computation, we amortize the per-vector miss rate over all streams. If we assume that none of the banks is on the correct page when the MSU changes FIFOs, then this average is $R = \frac{v}{s} \times r$. But if:

1) we assume a mode of SMC operation in which the MSU takes turns servicing each FIFO, providing as much service as possible before moving on to service another FIFO;

2) the MSU has filled all the FIFOs and must wait for the CE to drain them before issuing more accesses; and

3) the first FIFO to be serviced during the next “turn” was the last to be serviced during the previous one,

then the MSU need not pay the DRAM page-miss overhead again at the beginning of the next turn. Thus the MSU may avoid paying the per-bank page-miss overhead for one vector at each turn. When we exploit this phenomenon, our average page-miss rate becomes:

$$R = \frac{v-1}{s} \times r = \frac{v-1}{s} \times \frac{b\,(s-1)}{gcd\,(b, stride) \times fs} = \frac{b\,(s-1)\;(v-1)}{gcd\,(b, stride) \times fs^2} \qquad (5)$$

Let *h* be the cost of servicing an access that hits the current DRAM page, and let *m* be the cost of servicing an access that misses the current DRAM page. The maximum achievable bandwidth for a computation is equal to the percentage of banks used, thus we must scale our bandwidth formula accordingly, dividing by the greatest common denominator of the total number of banks and the vector stride. The percentage of peak bandwidth for the computation is thus:

$$\%\text{ peak bandwidth} = \frac{h}{(R \times m) + ((1-R) \times h)} \times \frac{100.0}{gcd\,(b, stride)} \qquad (6)$$

Single-Vector and Large-Stride Computations

Note that for a computation involving a single vector, only the first access to each bank generates a DRAM page miss. If we maintain our assumption that pages are infinitely large, all remaining accesses will hit the current page. In this case, our model produces a page-miss rate of 0, and the predicted percentage of peak bandwidth is 100. We can more accurately bound performance by considering the actual number of data elements in a page and calculating the precise number of page-misses that the computation will incur.

Likewise, for computations involving vectors with large strides, the predominant factor affecting performance is no longer FIFO depth, but the number of vector elements per page. The number of elements is the page size divided by the stride of the vector data within the

memory bank., and the distance between elements in a given bank is the vector stride divided by the number of banks the vector hits. We shall refer to this value as the *effective intrabank stride*, or *EIS*:

$$\text{EIS} = \frac{stride}{gcd\,(b, stride)} \tag{7}$$

Thus for a system with two interleaved banks, elements of a stride-two vector have an EIS of 1 and are contiguous within a single bank of memory.

Decreasing DRAM page size and increasing vector stride affect SMC performance in similar ways. Let *d* be the number of data elements in a DRAM page. Then for computations involving a single vector or multiple vectors with large EIS values, the average page-miss rate per FIFO is:

$$R = EIS/d \tag{8}$$

For single-vector computations or computations in which EIS/d is less than the FIFO depth, we use Equation 8 instead of Equation 5 to calculate R. The percentage of peak bandwidth is then calculated from Equation 6, as before. Note that neither FIFO depth nor the CE's access pattern affects performance for large-stride computations. Computations involving vectors with $EIS \geq d$ all have the same performance, since in these cases only one vector element resides in a DRAM page.

6.2 Multiprocessor Extensions

Given the similarity of the memory subsystems for the SMC organizations described in Section 2, we might expect an SMP SMC system to behave much like a uniprocessor SMC with a large number of FIFOs. For SMP systems, though, some of the assumptions made in the uniprocessor performance models no longer hold. For instance, we can no longer assume that each single-access vector occupies only one FIFO (or, equivalently, that each double-access vector occupies two). As we saw in Section 3, the distribution of vectors

among the FIFOs depends upon how the workload is parallelized. The parallelization scheme affects the CEs' pattern of DRAM page-sharing, which in turn affects performance.

We can bound SMP SMC performance for both prescheduled and statically scheduled workloads by calculating the minimum number of page misses for the extreme case when *all* CEs share the same DRAM pages. We could also compute a very conservative estimate of performance by calculating the maximum percentage of peak achievable when *no* CEs share DRAM pages at any point in the computation.

Recall that the system is balanced so that if each of N CEs can consume a data item each cycle, the memory system provides enough bandwidth to perform N near accesses in each processor cycle. Each CE can only consume data from its set of FIFOs, while the MSU may arrange for all accesses to be for a single FIFO at a time: this means that the memory system can now fill a FIFO N times faster. Let M be the number of CEs participating in the computation. When all CEs use the same DRAM pages, we have essentially distributed each of our s streams over M FIFOs, a situation that is analagous to using a single FIFO of depth $F = M \times f$ for each stream.

We assume a model of computation in which each CE accesses its FIFOs in round-robin order, consuming one data item from a FIFO at each access. It takes the MSU F/N cycles to supply F items for a stream. During this time, each CE will consume $\frac{F}{Ns}$ more data elements from this stream, for a total of $\frac{MF}{Ns}$ freed FIFO positions. While the MSU is filling those FIFO positions (in $\frac{MF}{N^2 s}$ cycles), the CE can remove $\frac{M^2 F}{N^2 s^2}$ more, and so on. Thus our model for calculating the page-miss rate of a stream becomes:

$$r = \frac{b}{gcd\,(b, stride)} \times \frac{1}{F\left(1 + \frac{M}{Ns} + \left(\frac{M}{Ns}\right)^2 + \left(\frac{M}{Ns}\right)^3 + \ldots\right)} \tag{9}$$

Our equation for the average page-miss rate is now:

$$R = \frac{v-1}{s} \times r = \frac{v-1}{s} \times \frac{b\,(Ns-M)}{gcd\,(b, stride) \times FNs} = \frac{b\,(Ns-M)\;(v-1)}{gcd\,(b, stride) \times FNs^2} \qquad (10)$$

And the percentage of peak bandwidth is computed as in Equation 6:

$$\%\ \text{peak bandwidth} = \left(\frac{h}{(R \times m) + ((1-R) \times h)} \times \frac{100.0}{gcd\,(b, stride)} \right)$$

When the data being used by any particular CE occupies different DRAM pages from that used by the other CEs, it is as if the CE were processing its own distinct set of vectors. This situation arises frequently for prescheduled workloads. We can compute a better performance estimate if we take into account the different page-sharing patterns encountered during the course of the computation, adjusting the number of vectors and streams accordingly. For instance, if we draw a vertical line at each of the page boundaries in Figure 5, we will have divided the computation into three distinct phases, each having a different page-sharing pattern. If we then assume that all CEs proceed at approximately the same rate — that is, if we assume that the *spatial* divisions of data correspond to *temporal* phases of the computation — we can then apply the asymptotic model to each phase, computing the overall percentage of peak bandwidth as a weighted average of the maximum performance for each phase.

7. Simulation Environment

In order to validate the SMC concept, we have simulated a wide range of SMC configurations and benchmarks, varying FIFO depth, dynamic order/issue policy, number of CEs, number of memory banks, DRAM speed, benchmark kernel, and vector length, stride, and alignment with respect to memory banks. Complete uniprocessor results, including a detailed description of each access-ordering heuristic, can be found in [McK93a]; highlights of these results are presented in [McK94a, McK94b]. Complete shared-memory multiprocessor results can be found in [McK94c]. Since our concern here is to correlate the performance bounds of our analytic model with our functional simulation

results, we present only the maximum percentage of peak bandwidth attained by any order/issue policy simulated for a given memory system and benchmark. All simulation results here were generated using DRAM pages of 4K bytes.

8. Benchmark Suite

Many types of applications are limited by memory bandwidth, including scientific computations. Caching may provide adequate bandwidth for some, but not all, portions of such programs. The bottlenecks in these computations usually take the form of memory-intensive inner loops that make linear traversals of vector-like data. Each element is typically visited only once during lengthy portions of the computation, and this lack of temporal locality of reference makes caching less effective.

We have chosen a suite of benchmark kernels representing access patterns found in real scientific codes. Listed in Figure 9, this suite constitutes a representative subset of all *possible* access patterns for computations involving a small number of vectors. Recall that in order to put as much stress as possible on the memory system, we model the processor as a generator of non-cached loads and stores of vector elements. Scalar and instruction references are assumed to hit in the cache, and all stream references use non-caching loads and stores. The *hydro* and *tridiag* kernels share the same access pattern, thus their results for our models and simulations are identical, and are presented together.

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:	$\forall i$	$tmp \leftarrow y_i \qquad y_i \leftarrow x_i \qquad x_i \leftarrow tmp$
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 9 Benchmark Kernels

9. Results

Figure 10 through Figure 15 depict performance for stride-one vectors as a function of FIFO depth and the number of memory banks (available concurrency). The results in Figure 10 and Figure 11 depict uniprocessor performance; those in Figure 12 and Figure 14 reflect SMP performance when prescheduling is used to parallelize the computation; and those in Figure 13 and Figure 15 illustrate SMP performance when static scheduling is used. Figure 16 illustrates how our performance limits change as vector stride increases, and Figure 17 depicts performance when not all CEs are used in a computation.

All results are given as a percentage of the system's peak bandwidth, the bandwidth necessary to allow the CEs to perform a memory operation each cycle. The vectors used for these simulations are 100, 10,000, and 80,000 doublewords in length. Given the overwhelming similarity of the performance trends for most benchmarks and system configurations, we only discuss highlights of our results here. Other results can be found in the Appendix, and a more detailed comparison of uniprocessor simulation and asymptotic model results can be found in [McK93b, McK93c].

Figure 10 depicts results for a uniprocessor SMC system with one bank. The vectors for Figure 10(a) and Figure 10(b) are 100 elements long; those for Figure 10(c) and Figure 10(d) are 10,000 elements. Figure 10(a) and Figure 10(c) illustrate the performance curves for *vaxpy*, which involves three vectors: a vector *a* times a vector *x*, plus a vector *y*. Figure 10(b) and Figure 10(d) show results for *scale*, our single-vector kernel.

For multiple-vector computations on short vectors, the startup-delay bound is the limiting performance factor, as evidenced by the curves in Figure 10(a). Short vectors prevent the SMC from effectively amortizing both the startup costs and DRAM page-miss overheads. Note that performance is constant for FIFO depths greater than the vector length. For longer vectors, as in Figure 10(c), startup-delays cease to impose significant limits to achievable

bandwidth, and simulation performance approaches the asymptotic bound. The performance effects of different vector lengths can also be seen for the *scale* benchmark depicted in Figure 10(b) and Figure 10(d) — in both cases, the theoretical performance limits are barely below 100% of peak, yet SMC simulation performance on shorter vectors is a few percent of peak lower than for vectors of length 10,000.

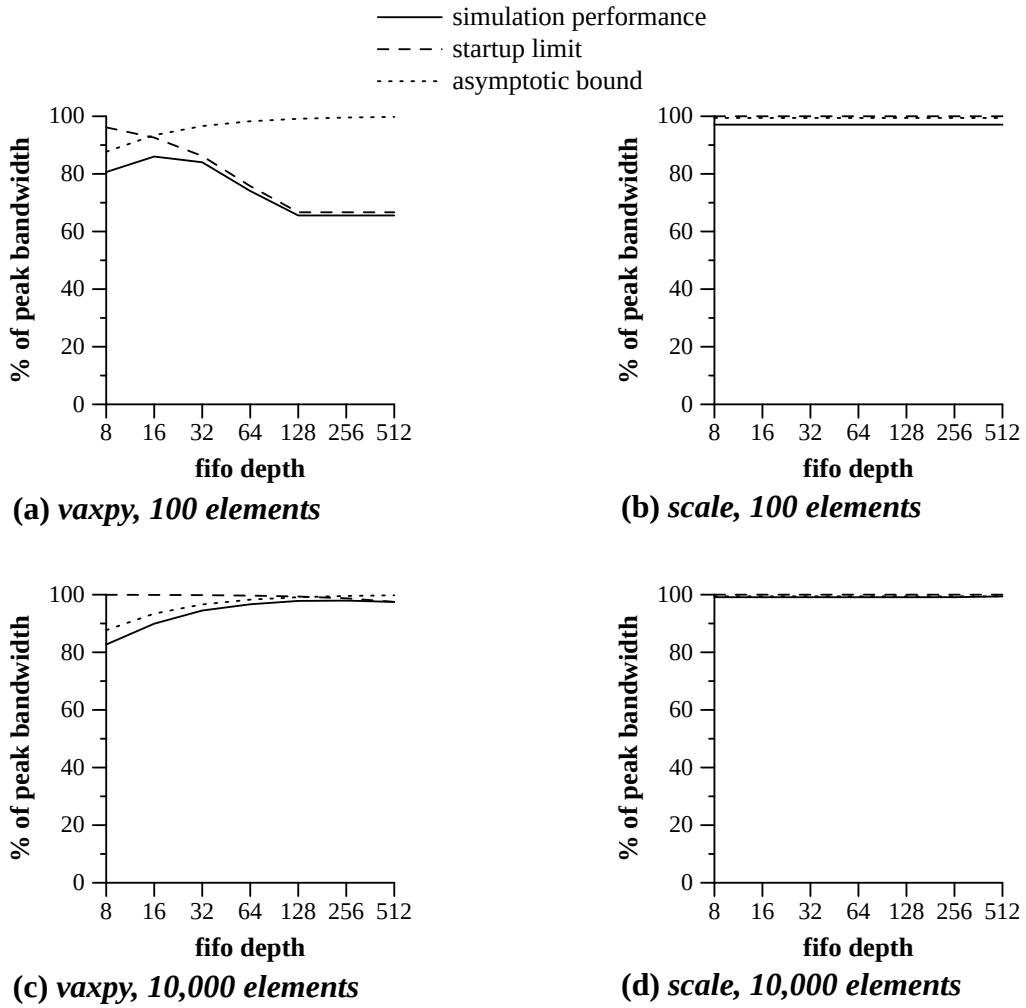
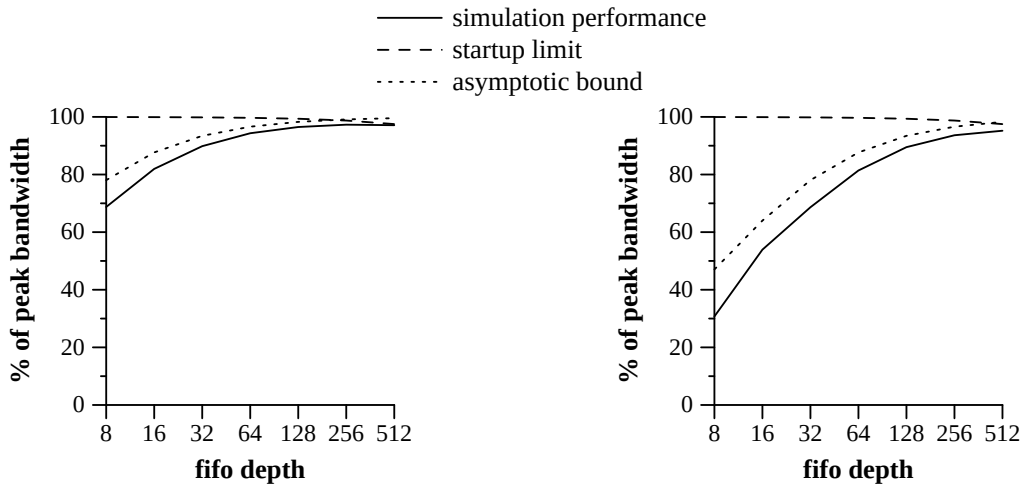


Figure 10 Performance for a Uniprocessor with 1 Memory Bank

If we increase the number of memory banks, we decrease the number of vector elements in each bank: doubling the number of memory banks has a similar effect on performance as halving the vector length. Decreasing the number of elements per bank limits the SMC's ability to amortize page-miss and startup costs, thus performance for systems with many

banks is farther from the asymptotic limits than for a system with fewer banks. Note that systems with more banks deliver a smaller percentage of a much *greater* bandwidth. Figure 11 demonstrates this phenomenon for uniprocessor systems with 2 and 8 banks.

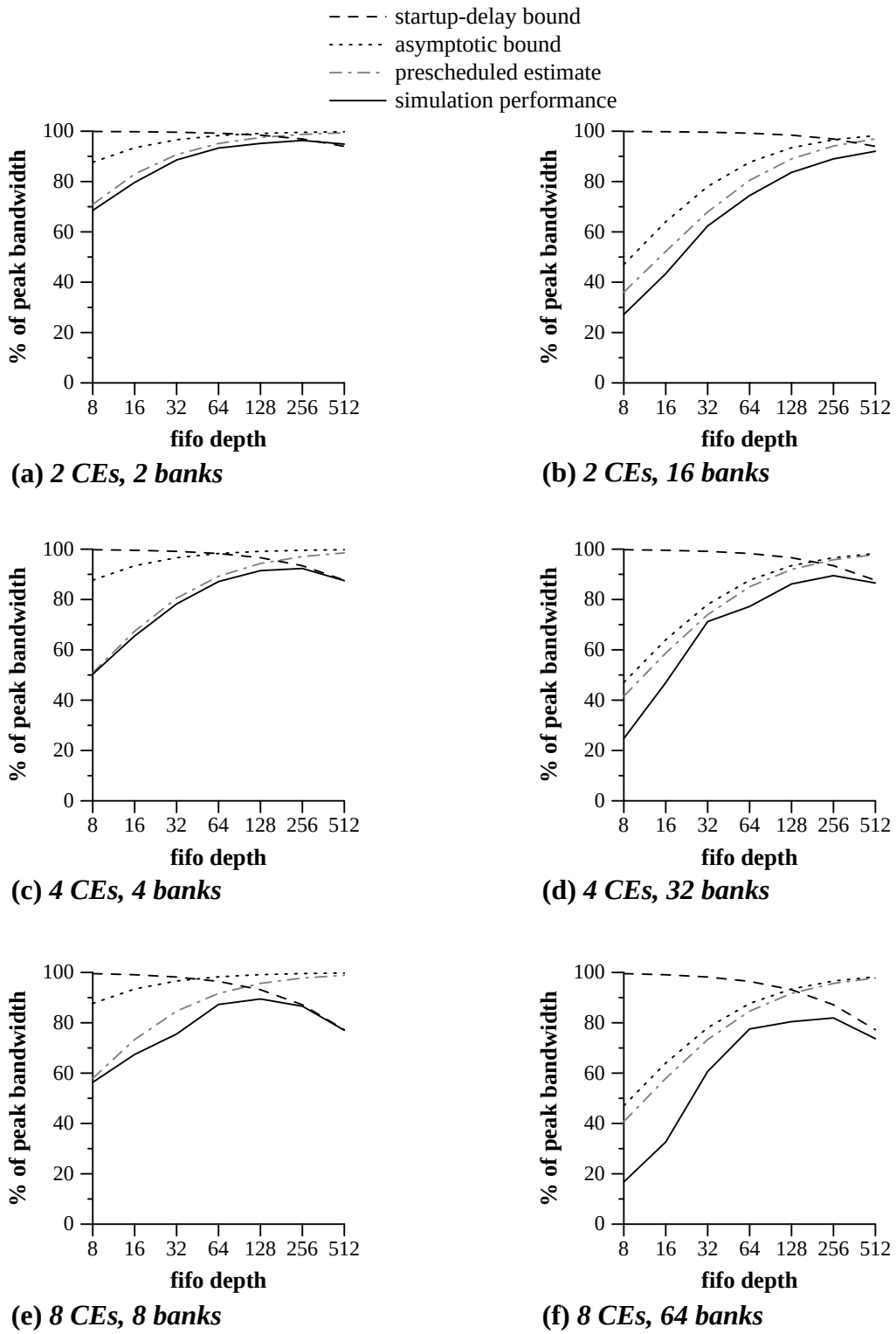


(a) *vaxpy, 2 banks* **(b) *vaxpy, 8 banks***

Figure 11 Uniprocessor Performance for Increasing Banks

Recall that prescheduling breaks a vector into chunks, giving a different chunk to each CE. Using more CEs to execute a prescheduled workload results in shorter vectors at each CE, and as we saw in our uniprocessor results, shorter vectors limit the number of accesses over which startup and page-miss costs can be amortized. Figure 12 depicts results for the *vaxpy* benchmark on systems with 2, 4, and 8 CEs. The asymptotic bounds depicted here are calculated assuming the best-case DRAM page-sharing scenario among the CEs: all CEs are assumed to use the same working set of DRAM pages throughout the computation. In actuality, this is rarely the case for prescheduled workloads, but the assumption allows us to calculate true performance bounds. The startup-delay bounds in all our SMP graphs were calculated using Equation 3 from Section 5.

Although it is unlikely that system designers would build an SMP system with a FIFO depth less than the number of memory banks, we include results for such systems for completeness and for purposes of comparison.



(a) ***2 CEs, 2 banks*** **(b)** ***2 CEs, 16 banks***

(c) ***4 CEs, 4 banks*** **(d)** ***4 CEs, 32 banks***

(e) ***8 CEs, 8 banks*** **(f)** ***8 CEs, 64 banks***

Figure 12 Prescheduled *vaxpy* for 10,000-Element Vectors

In addition to the asymptotic and startup-delay bounds, the graphs in Figure 12 include a grayed line indicating the performance estimate obtained by averaging the performance estimates of each page-sharing phase of the computation. This estimate incorporates the same assumption that is built into our functional simulations, namely that all CEs proceed at the same pace. Even though this curve does not represent a true bound, we include it to illustrate the distance between expected and maximal performance for shallow FIFOs. Note that the two curves converge as FIFO depth increases: at a depth of 512, the bound exceeds the estimate by only a few percent of peak.

Figure 13 depicts performance when static scheduling is used to parallelize a workload. These graphs compare the theoretical bounds to simulation results for *vaxpy* using vectors of length 10,000 on systems with 2 to 8 CEs. The startup-delay bound for this kind of workload is computed using Equation 1, just as for a uniprocessor system. The asymptotic limit for static scheduling is identical to that for the prescheduling, but because statically scheduled computations tend to enjoy a higher degree of DRAM page-sharing, their simulation performance more closely approaches the bounds. This phenomenon is particularly evident in the performance curves for computations using 80,000-element vectors on the 8-CE systems of Figure 14 and Figure 15. For instance, on a 64-bank system, the prescheduled *vaxpy* computation of Figure 14(b) delivers 27% less of the peak system bandwidth than the statically scheduled computation of Figure 15(b). The latter curve comes within 3% of peak of the asymptotic limit for a FIFO depth of 256.

As the number of CEs in the system increases and the amount of data processed by each CE decreases, performance is increasingly limited by the startup-delay bound. For instance, this bound starts to dominate performance at FIFO depths between 128 and 256 for the 2-CE systems of Figure 12 and Figure 13, but for the 8-CE system of Figure 12(e), the crossover point for the startup-delay and asymptotic limits is between FIFO depths of 32 and 64.

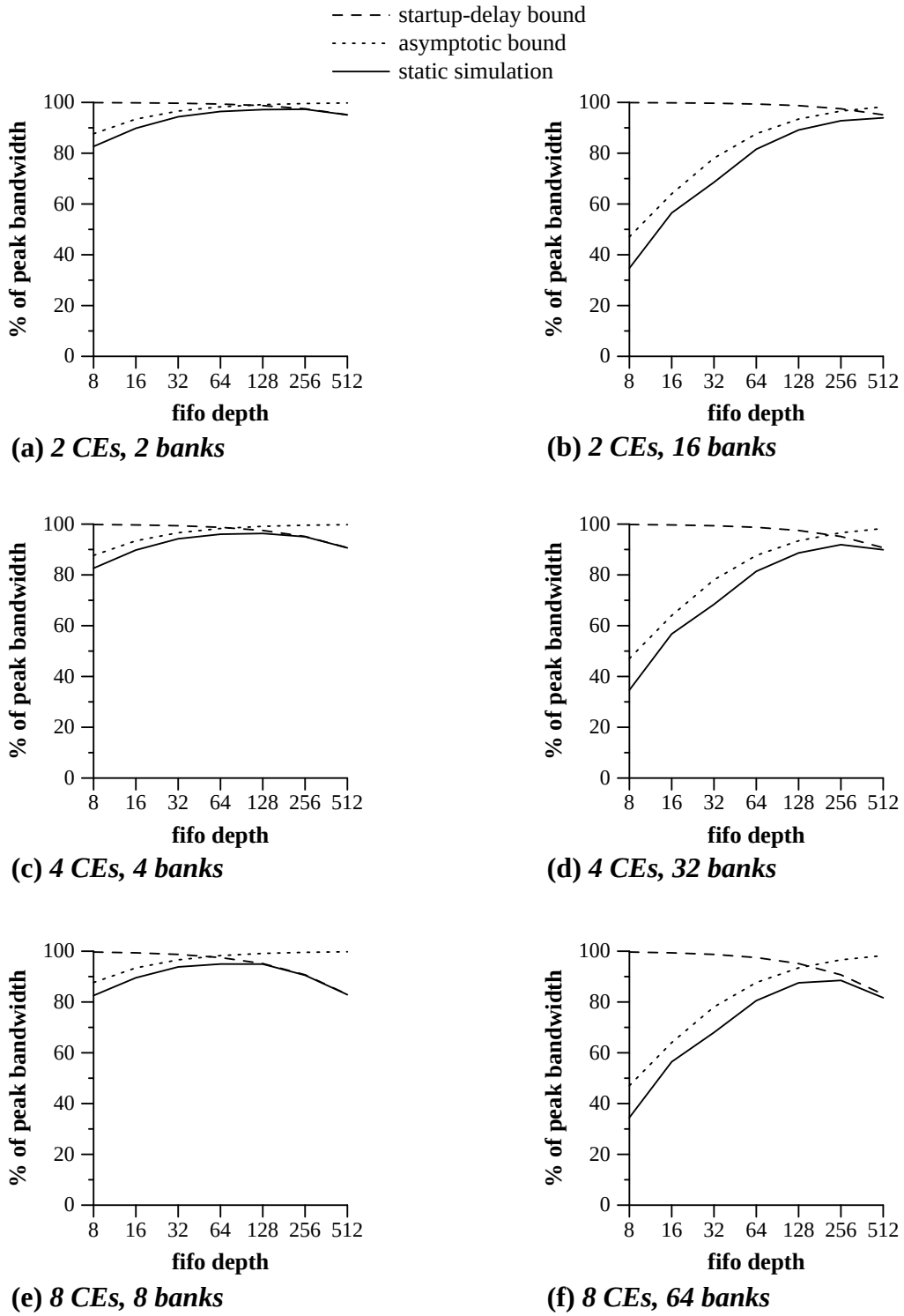
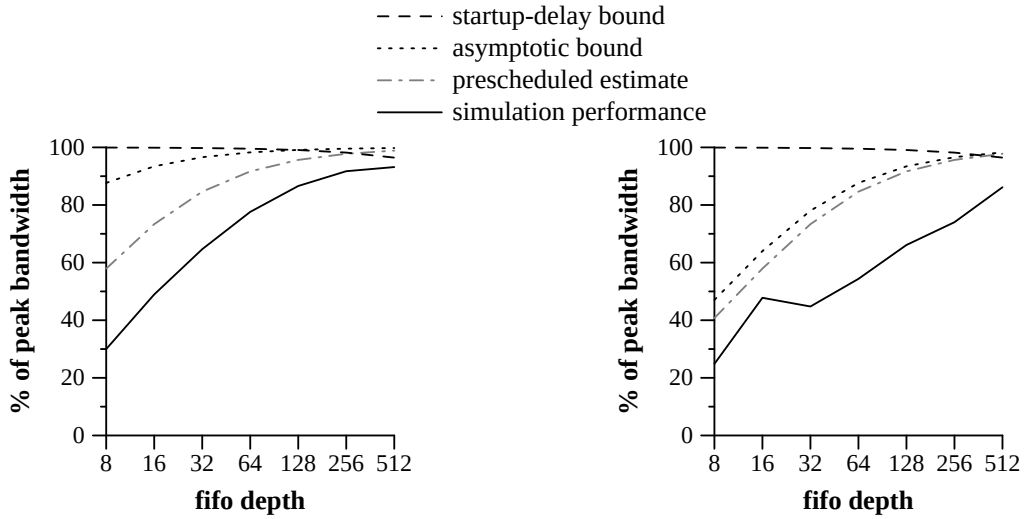


Figure 13 Statically Scheduled *vaxpy* for 10,000-Element Vectors

When prescheduling is used, the 10,000-element vectors of Figure 12 are too short for simulation performance to approach the asymptotic bounds. The 80,000-element vectors of Figure 14 allow the MSU to amortize page-miss costs over a greater number of fast accesses, thus performance for the 8-bank system in Figure 14(a) begins to approach the theoretical bounds for FIFO depths of at least 256. When we go to a system with 64 banks, however, even 80,000-element vectors fail to allow the MSU to effectively compensate for the page-miss overheads: performance in Figure 14(b) is uniformly lower than for the 10,000-element computation on the analogous system of Figure 12(f). The erratic shape of the simulation performance curve in Figure 14(b) results from using FIFO depths that are less than the number of memory banks; see the SMP SMC report for an explanation of the phenomenon [McK94c].



(a) *8 CEs, 8 banks* **(b) *8 CEs, 64 banks***

Figure 14 Prescheduled *vaxpy* for 80,000-Element Vectors

Figure 15 presents performance for 8-CE systems when static scheduling is used on computations with 80,000-element vectors. These performance curves are very similar to those for the analogous uniprocessor SMC systems: the high degree of DRAM page-sharing for this parallelization scheme yields roughly constant performance for a given ratio of CEs to number of banks and a given amount of data to be processed by each CE.

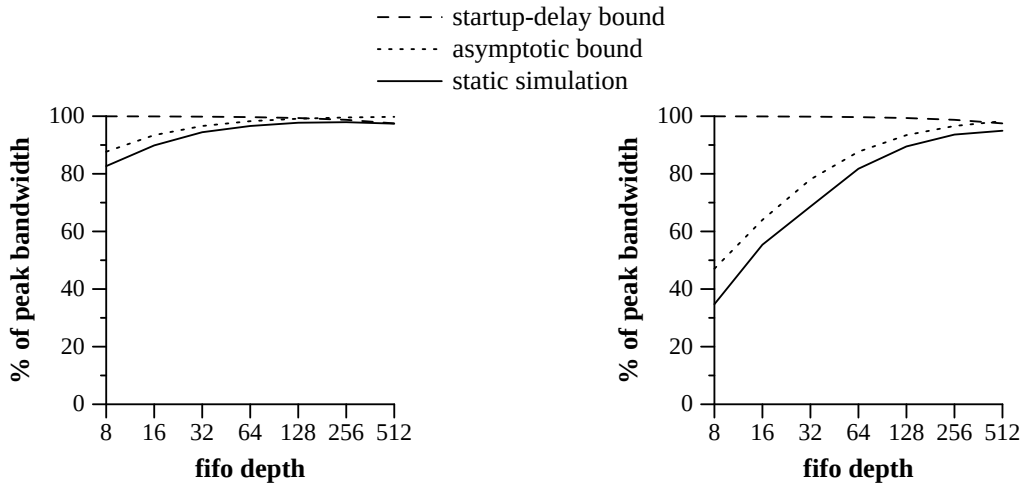


Figure 15 Statically Scheduled *vaxpy* for 80,000-Element Vectors

The graphs in Figure 10 through Figure 15 emphasize the importance of adjusting the FIFO depth to the computation. Deeper FIFOs do not always result in a higher percentage of peak bandwidth: for good performance, FIFO depth *must* be adjustable at runtime. Compilers can use the models presented here to calculate the optimal depth. This requires determining where the two limits intersect and selecting the next largest FIFO depth.

All examples thus far have been for vectors of stride 1, but the same performance limits apply for any small stride that is relatively prime to the number of banks. Strides that are not relatively prime prevent us from exploiting the full system bandwidth: whereas attainable bandwidth for unit-stride vectors is 100% of the system's total, the maximum for stride-two vectors is only 50% of peak. Figure 16 illustrates simulation results and performance limits for increasing strides. These results were generated using 10,000-element vectors and a 2-CE SMC system with 2 banks, a FIFO depth of 256, and DRAM pages of 4KB. We use the large-stride model (Equation 10 and Equation 6 from Section 6) to compute the asymptotic limits, since for these system parameters and strides the number

of elements residing in a DRAM page is never larger than the FIFO depth. Performance is constant for strides of 1024 or greater, since only one element resides in a DRAM page.

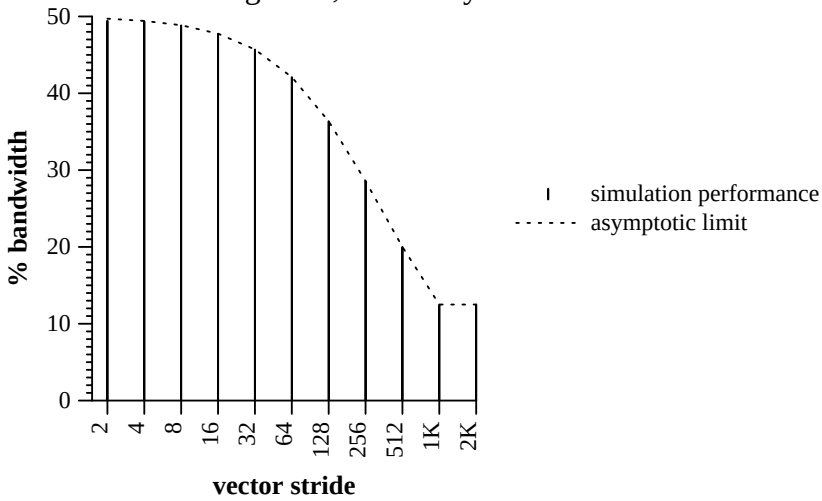
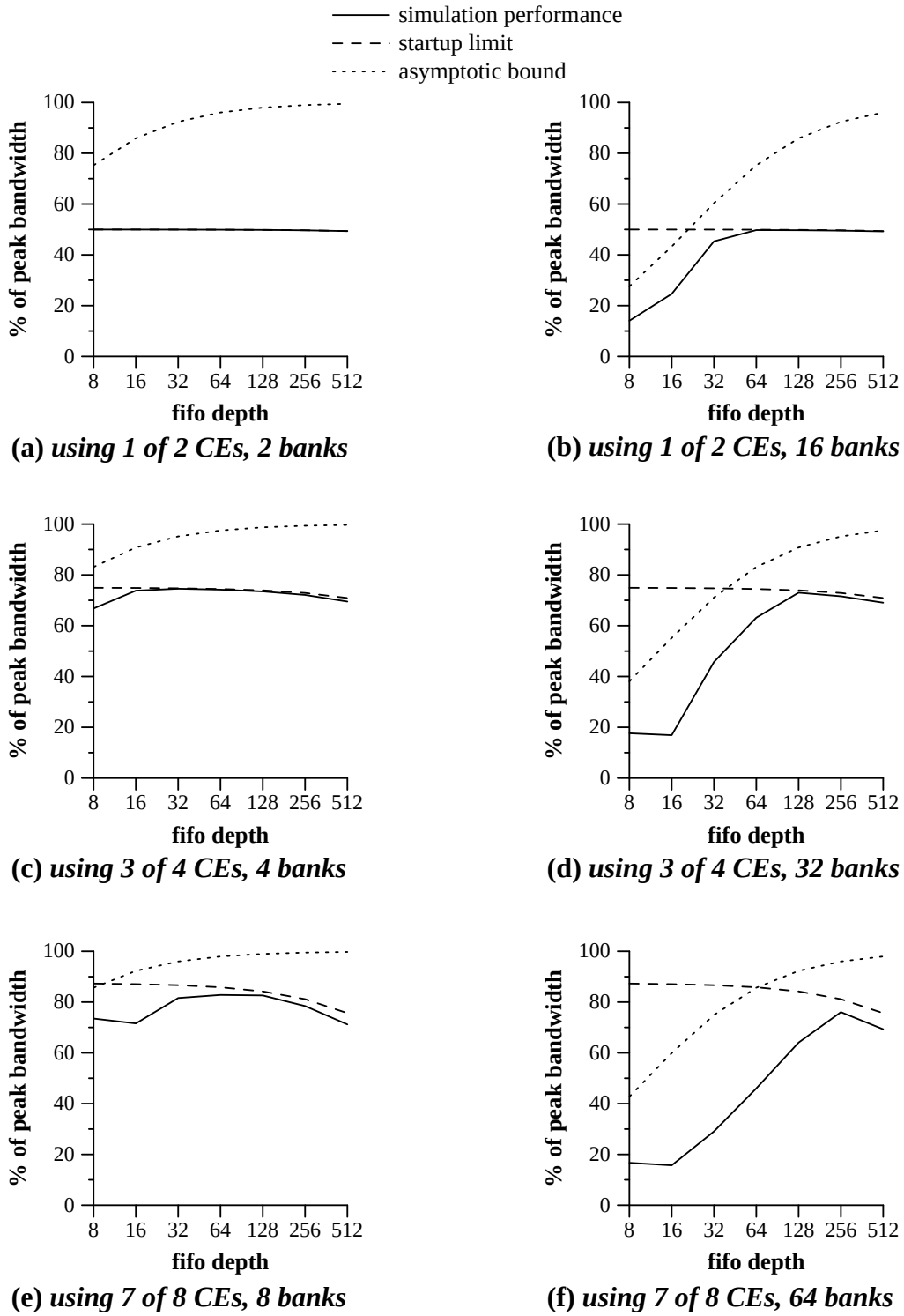


Figure 16 Asymptotic Limits for Increasing Strides

Figure 17 illustrates what happens when not all CEs participate in a computation. On SMC systems that only allow the MSU to service a single FIFO at a time, using fewer CEs may be an effective strategy for optimizing performance when using all CEs would yield an effective stride not relatively prime to the number of banks. For instance, by using one fewer CEs with static scheduling (in which each of M CEs performs every Mth loop iteration), the effective stride of the computation becomes relatively prime. The percentage of peak system bandwidth delivered becomes limited by the percentage of CEs used, rather than by the percentage of banks hit when all CEs are used. In such cases, the optimal FIFO depth to use is governed solely by the startup-delay bound.

10. 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.



(a) ***using 1 of 2 CEs, 2 banks*** **(b)** ***using 1 of 2 CEs, 16 banks***

(c) ***using 3 of 4 CEs, 4 banks*** **(d)** ***using 3 of 4 CEs, 32 banks***

(e) ***using 7 of 8 CEs, 8 banks*** **(f)** ***using 7 of 8 CEs, 64 banks***

Figure 17 Static *vaxpy* Performance for (N-1) CEs, 10,000-Element Vectors

Achieving greater bandwidth requires exploiting the characteristics of the *entire* memory hierarchy: it cannot be treated as if it were uniform access-time RAM. This knowledge should guide processor designs and operating system implementations, and 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 described our approach to dynamic access ordering, the Stream Memory Controller (SMC), and have presented analytic models to bound the performance of uniprocessor and symmetric multiprocessor SMC systems with memories comprised of page-mode DRAMs. Two different limits govern the percentage of peak bandwidth delivered:

- startup-delay bounds, or the amount of time a processor must wait to receive data for the first iteration of an inner loop; and
- asymptotic bounds, or the number of fast accesses over which the SMC can amortize DRAM page-miss costs.

The first limit bounds bandwidth utilization between the processors and the SMC; the second limit bounds the percentage of peak bandwidth exploited between the SMC and memory.

Our results emphasize one very important consideration in the design of any efficient SMC system: FIFO depth *must* be run-time selectable so that the amount of stream buffer space to use can be adapted to individual computations. Compilers can compute optimal depth using the equations presented here. Our analysis and simulation indicate that for sufficiently long vectors and appropriately deep FIFOs, SMC systems 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 grants MIP-9114110 and MIP-9307626.

Appendix

Figure 18 through Figure 60 depict analytic bounds and simulation performance for SMC systems with 1 to 8 computational elements. All results are given as a percentage of peak system bandwidth. The simulation results presented here represent the maximum bandwidth attained by any order/issue policy and vector alignment for stride-one vectors. The memory systems simulated consist of interleaved banks of 4K-byte, page-mode DRAMS.

Figure 18 through Figure 21 present uniprocessor SMC performance for our benchmark suite using vectors of length 100. Figure 22 through Figure 25 present results for the same systems and 10,000-element vectors. Figure 26 through Figure 29 illustrate performance on 2-CE systems when prescheduling is used to distribute vectors of length 10,000. Figure 30 through Figure 33 present prescheduled performance for 10,000-element vectors and 4-CE systems, and Figure 34 through Figure 37 depict results for the same computations on 8-CE systems. Figure 38 through Figure 41 present analogous information for 8-CE systems using vectors of length 80,000.

Figure 42 through Figure 57 present analogous results for statically scheduled computations on systems with 2, 4, and 8 CEs. Figure 58 through Figure 60 illustrate SMC performance when $N - 1$ CEs (where N is the total number of CEs in the system) are used with static scheduling.

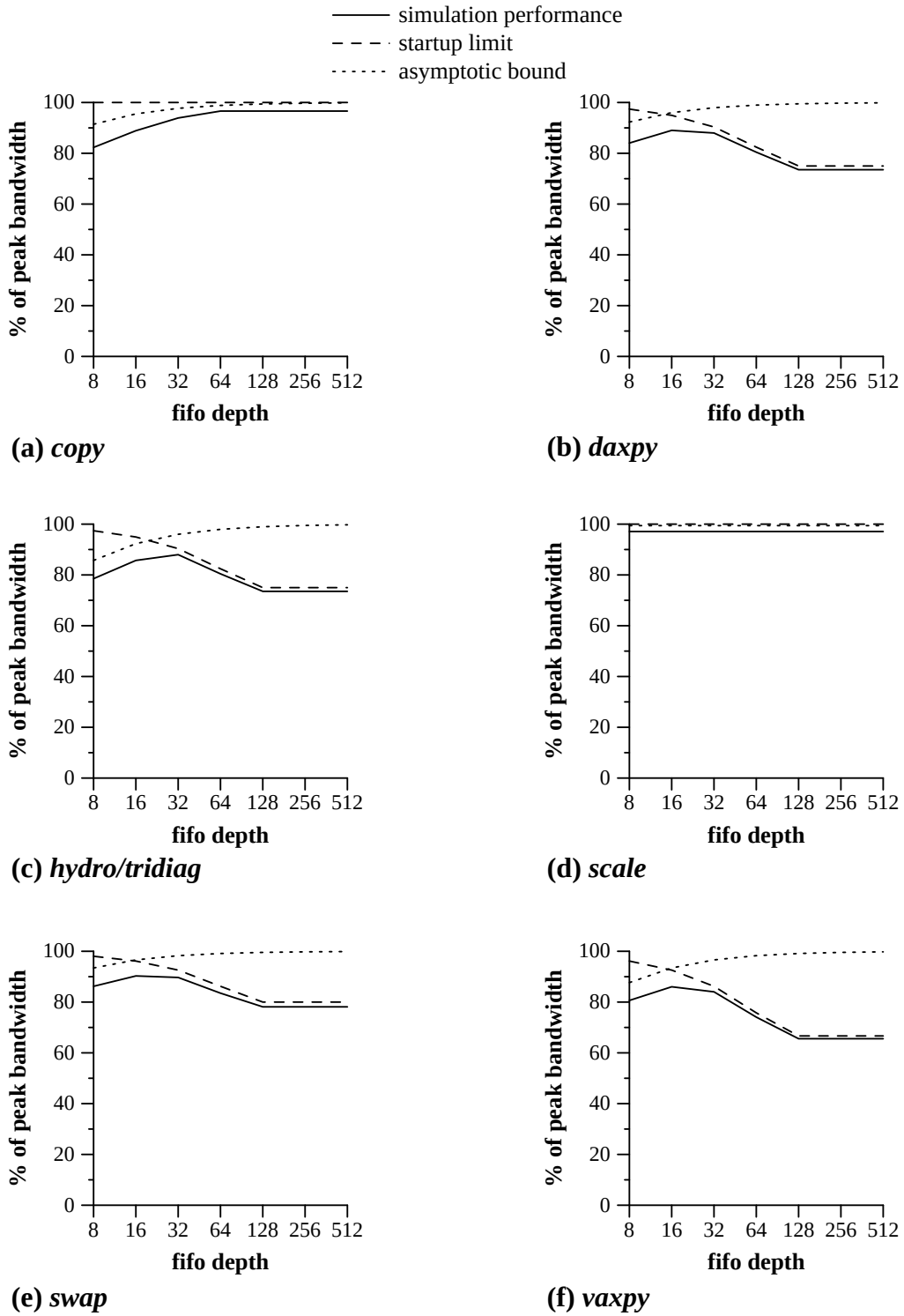


Figure 18 Presched Performance for 1 CE, 1 Bank, 100-Element Vectors

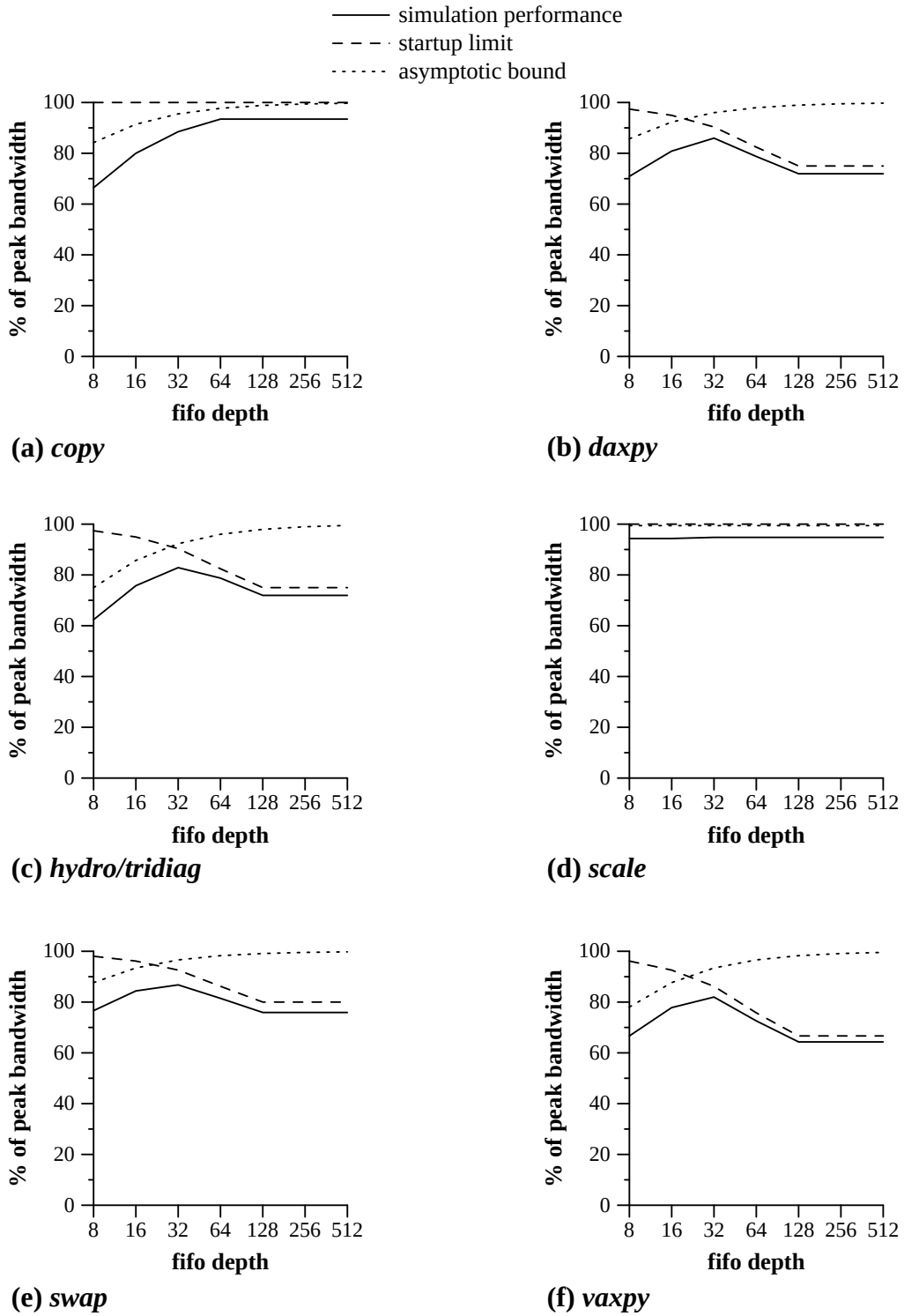


Figure 19 Presched Performance for 1 CE, 2 Banks, 100-Element Vectors

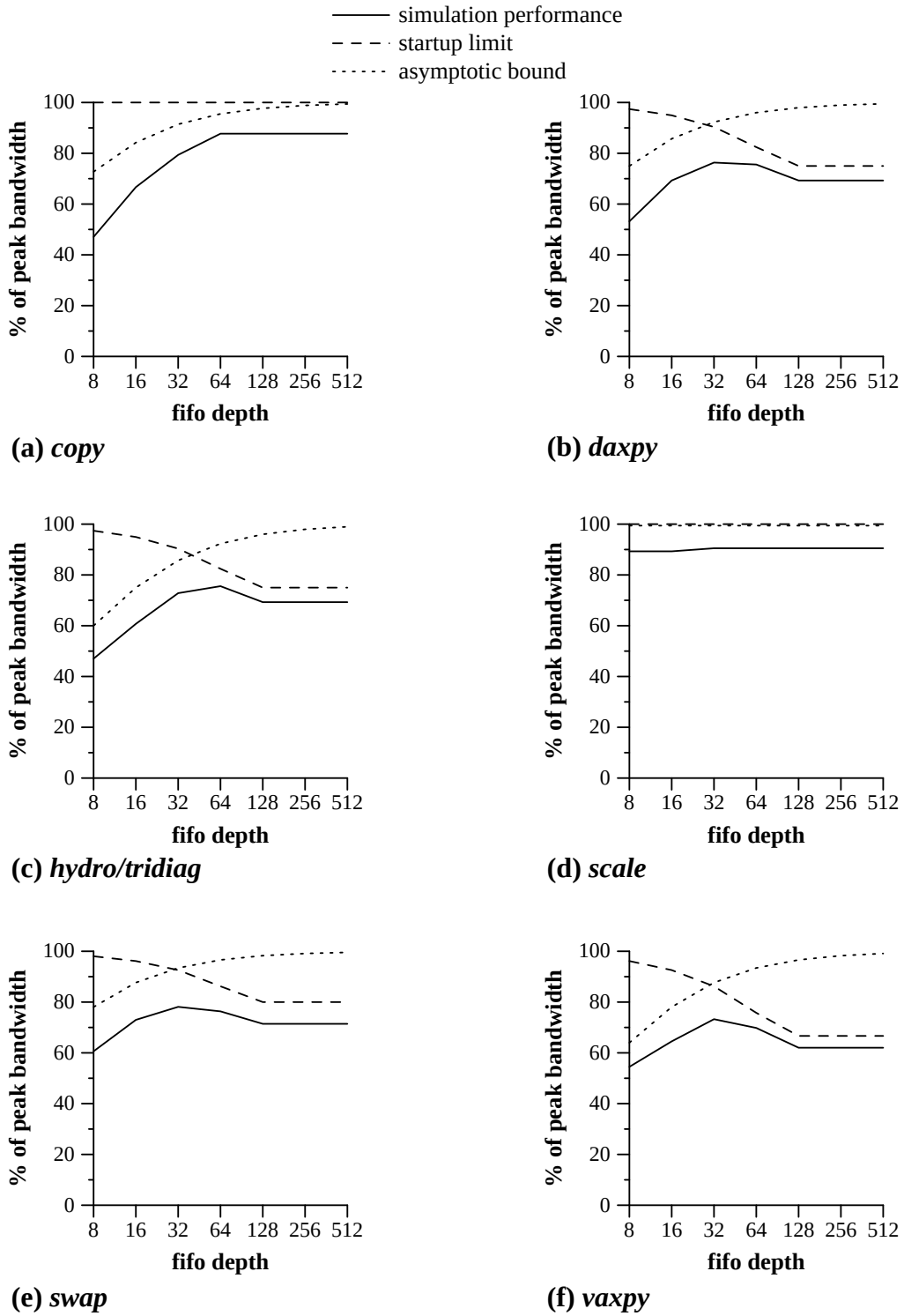


Figure 20 Presched Performance for 1 CE, 4 Banks, 100-Element Vectors

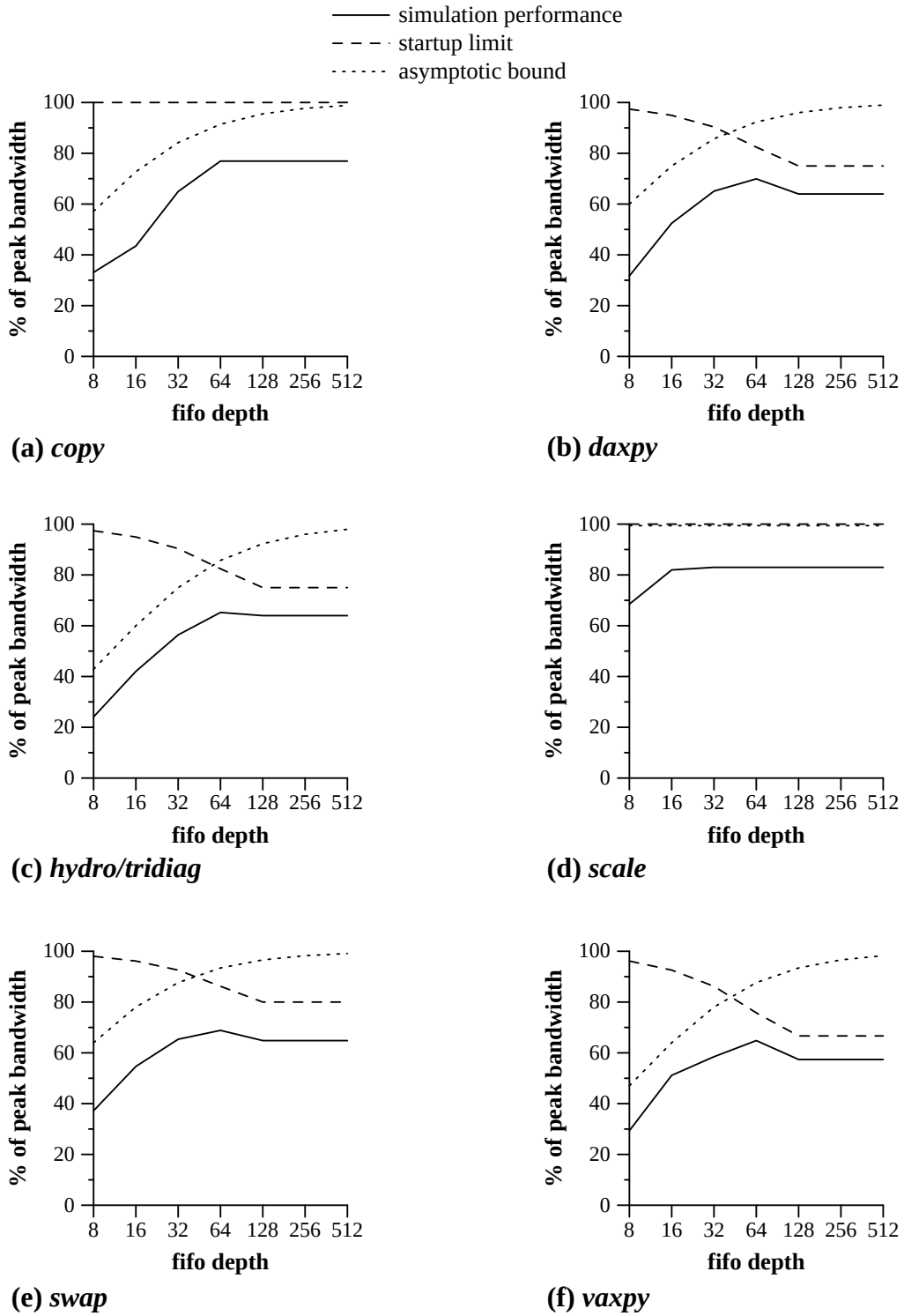


Figure 21 Presched Performance for 1 CE, 8 Banks, 100-Element Vectors

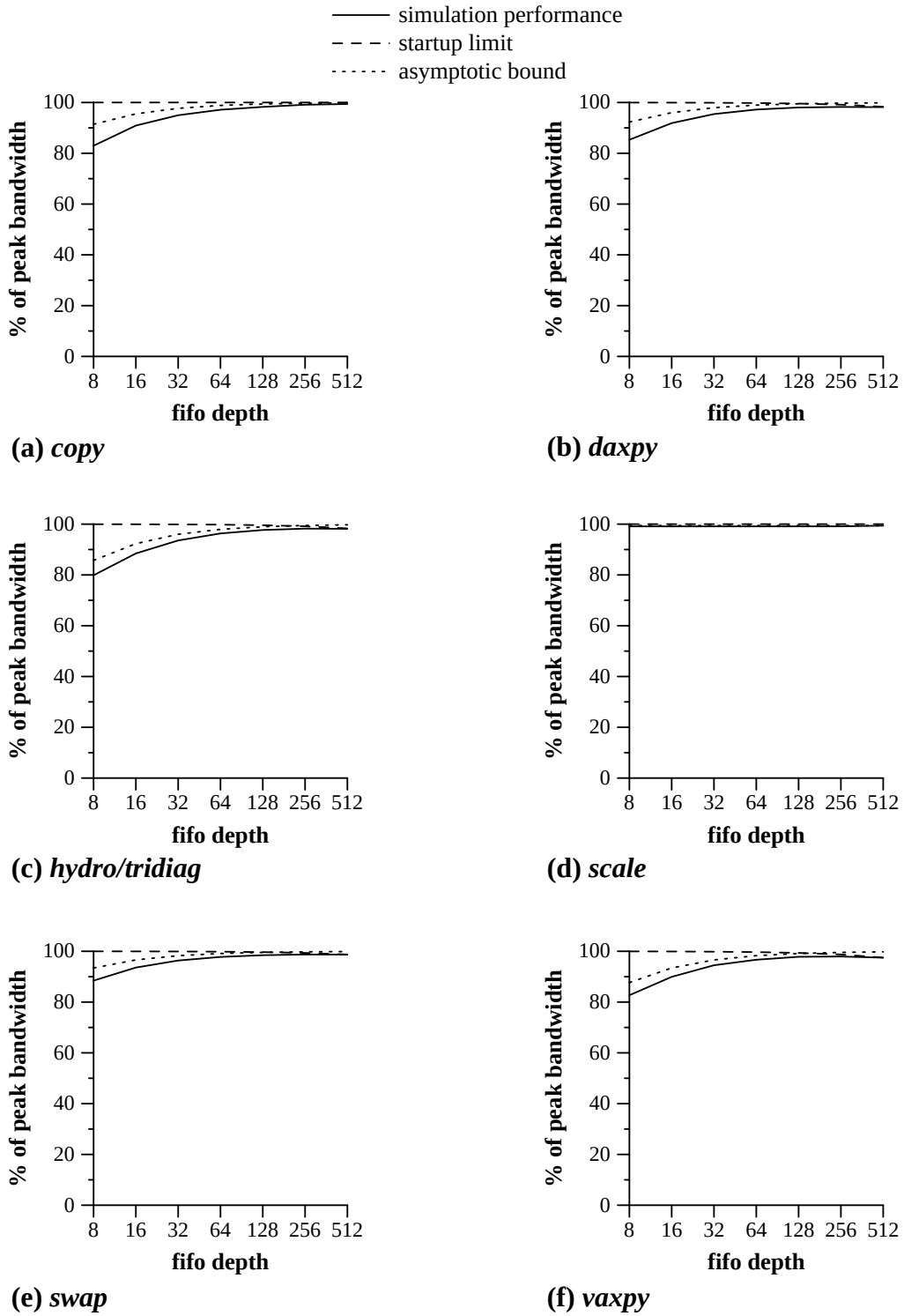


Figure 22 Presched Performance for 1 CE, 1 Bank, 10,000-Element Vectors

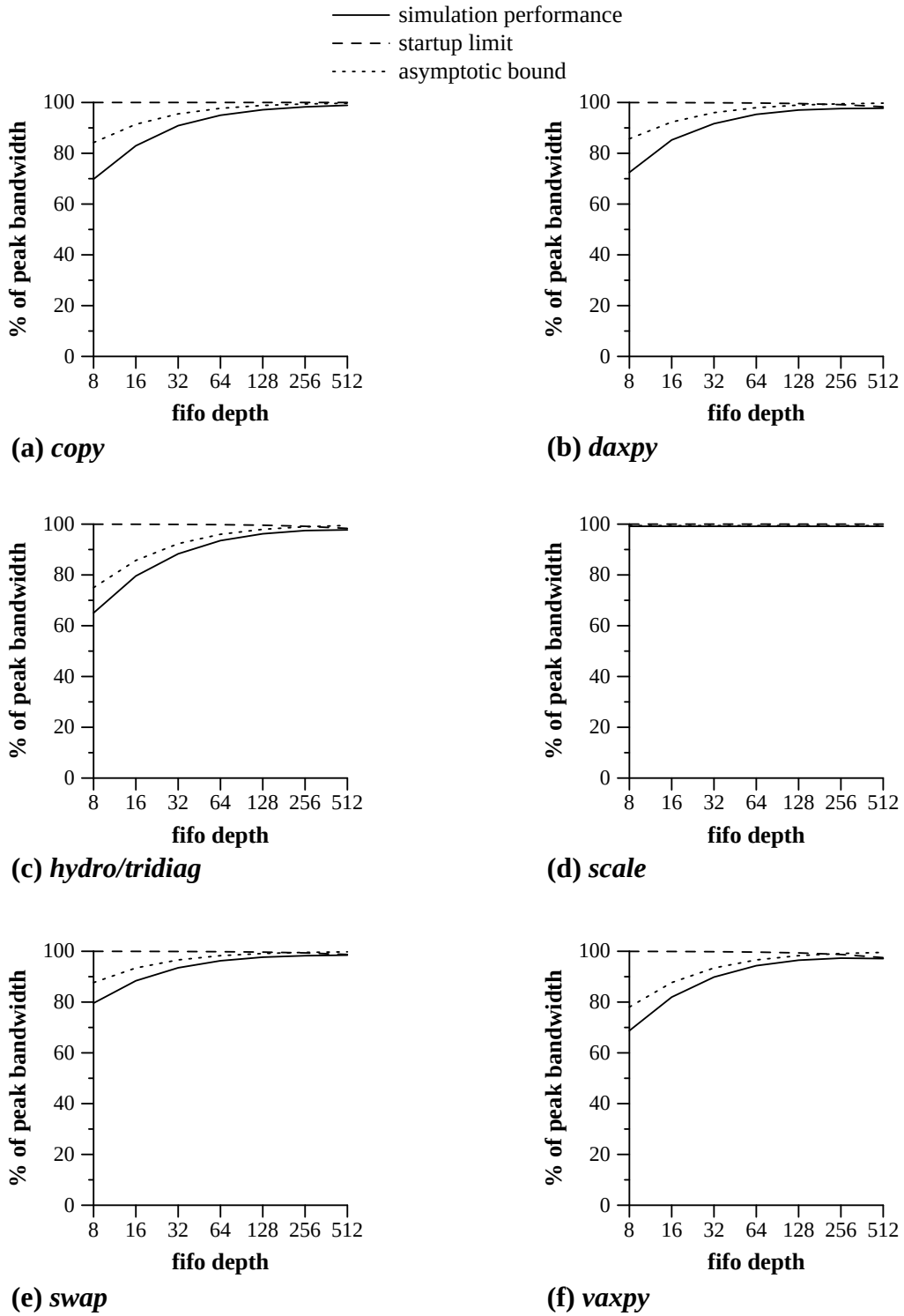


Figure 23 Presched Performance for 1 CE, 2 Banks, 10,000-Element Vectors

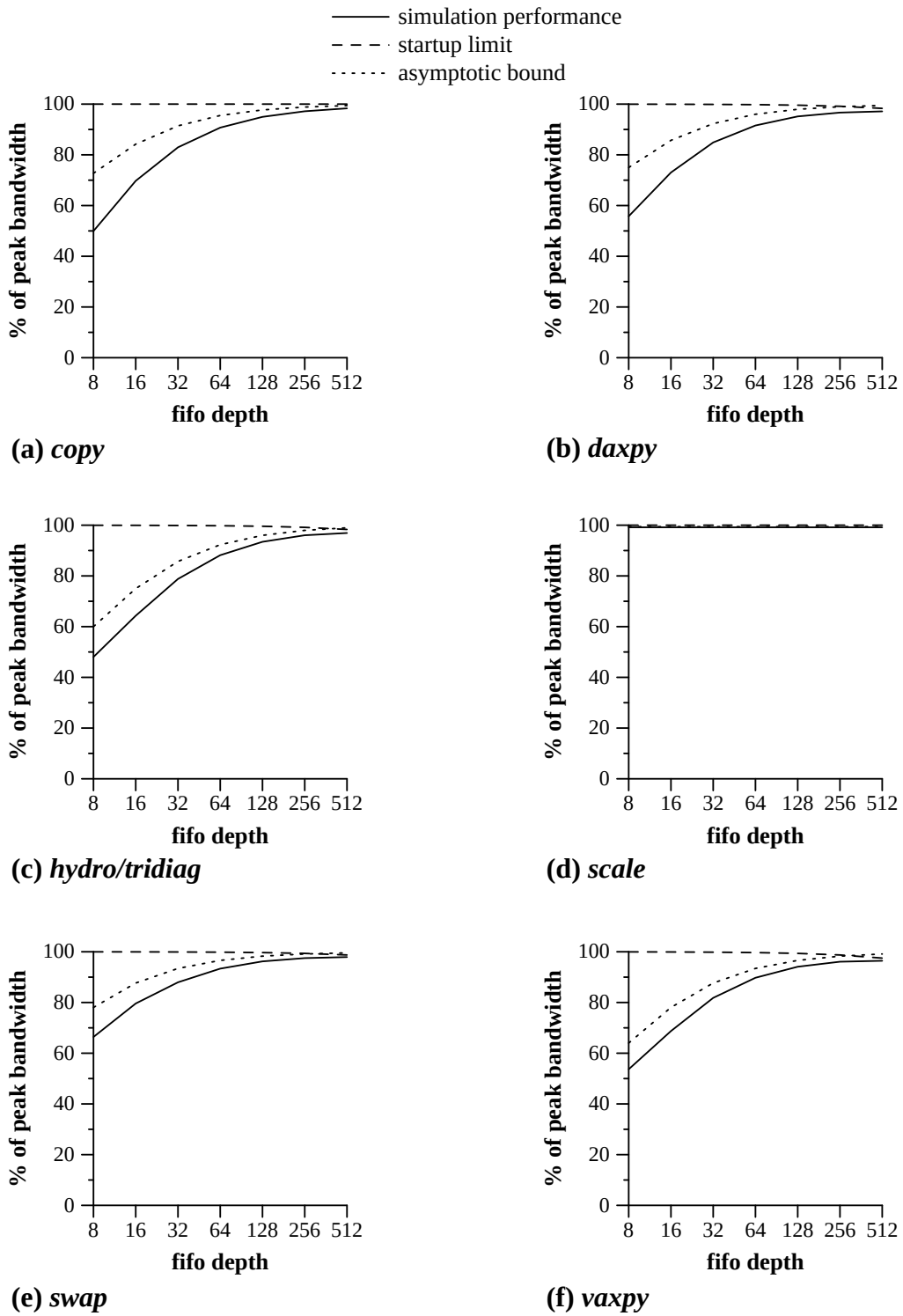


Figure 24 Presched Performance for 1 CE, 4 Banks, 10,000-Element Vectors

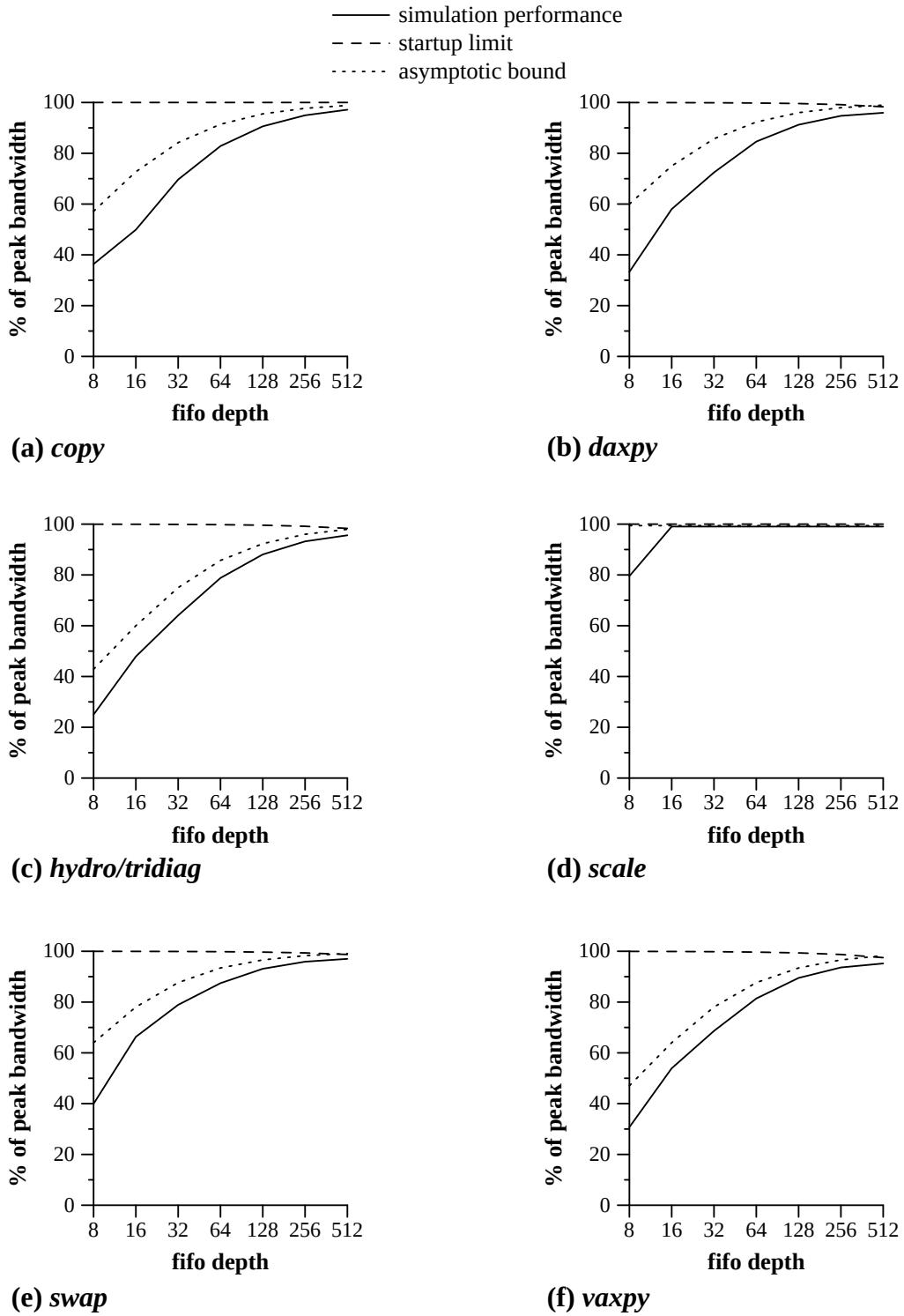


Figure 25 Presched Performance for 1 CE, 8 Banks, 10,000-Element Vectors

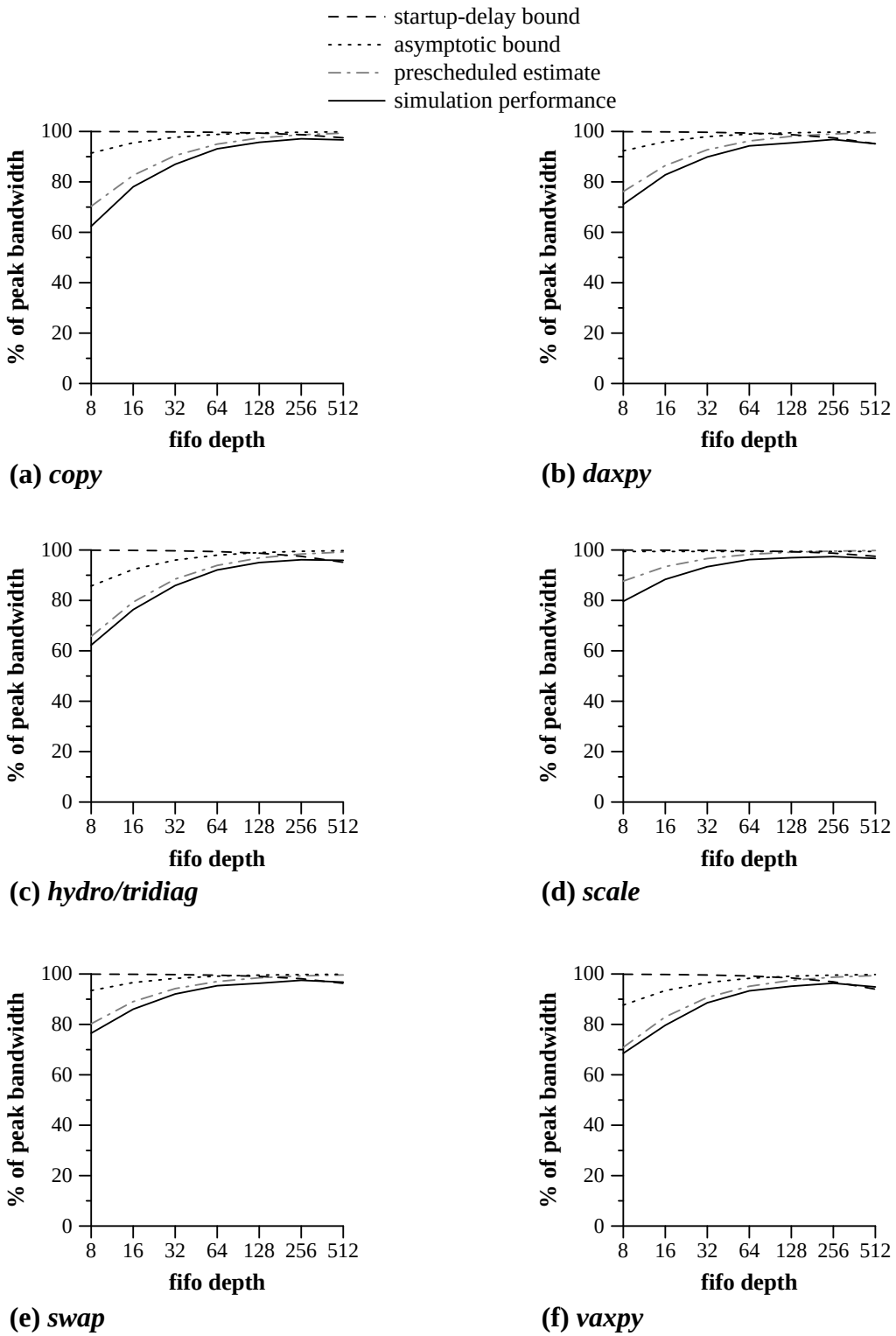


Figure 26 Presched Performance for 2 CEs, 2 Banks, 10,000-Element Vectors

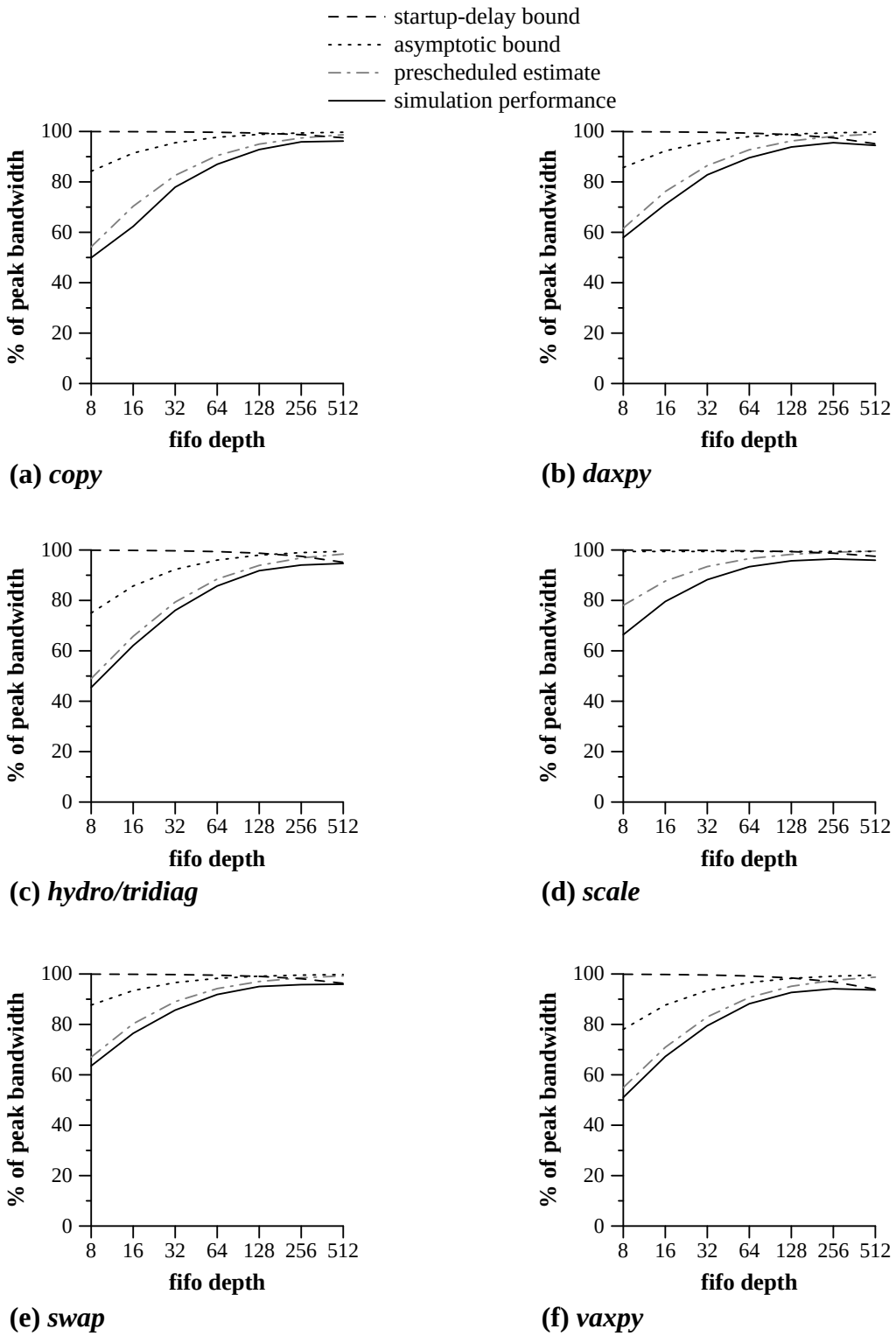


Figure 27 Presched Performance for 2 CEs, 4 Banks, 10,000-Element Vectors

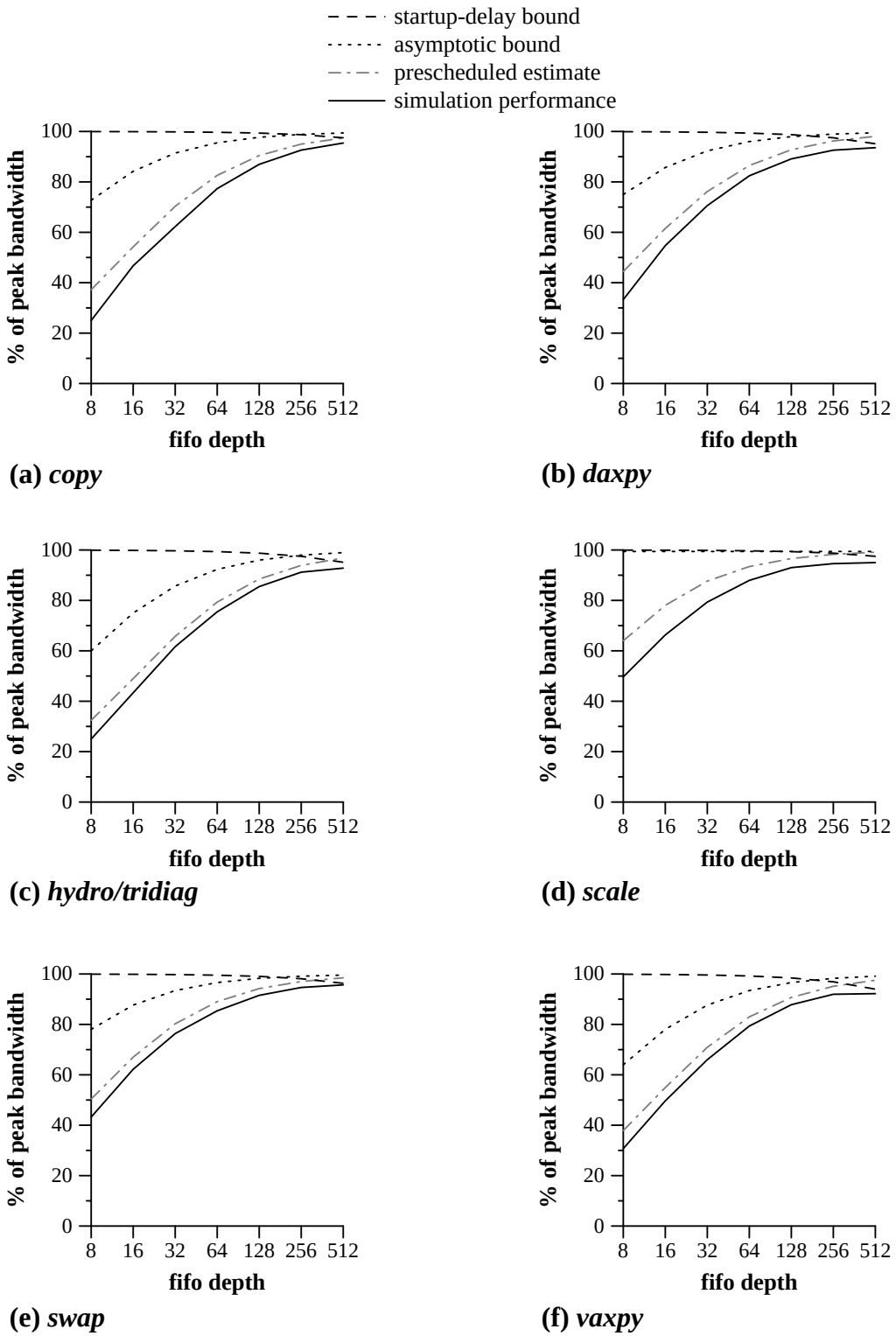


Figure 28 Presched Performance for 2 CEs, 8 Banks, 10,000-Element Vectors

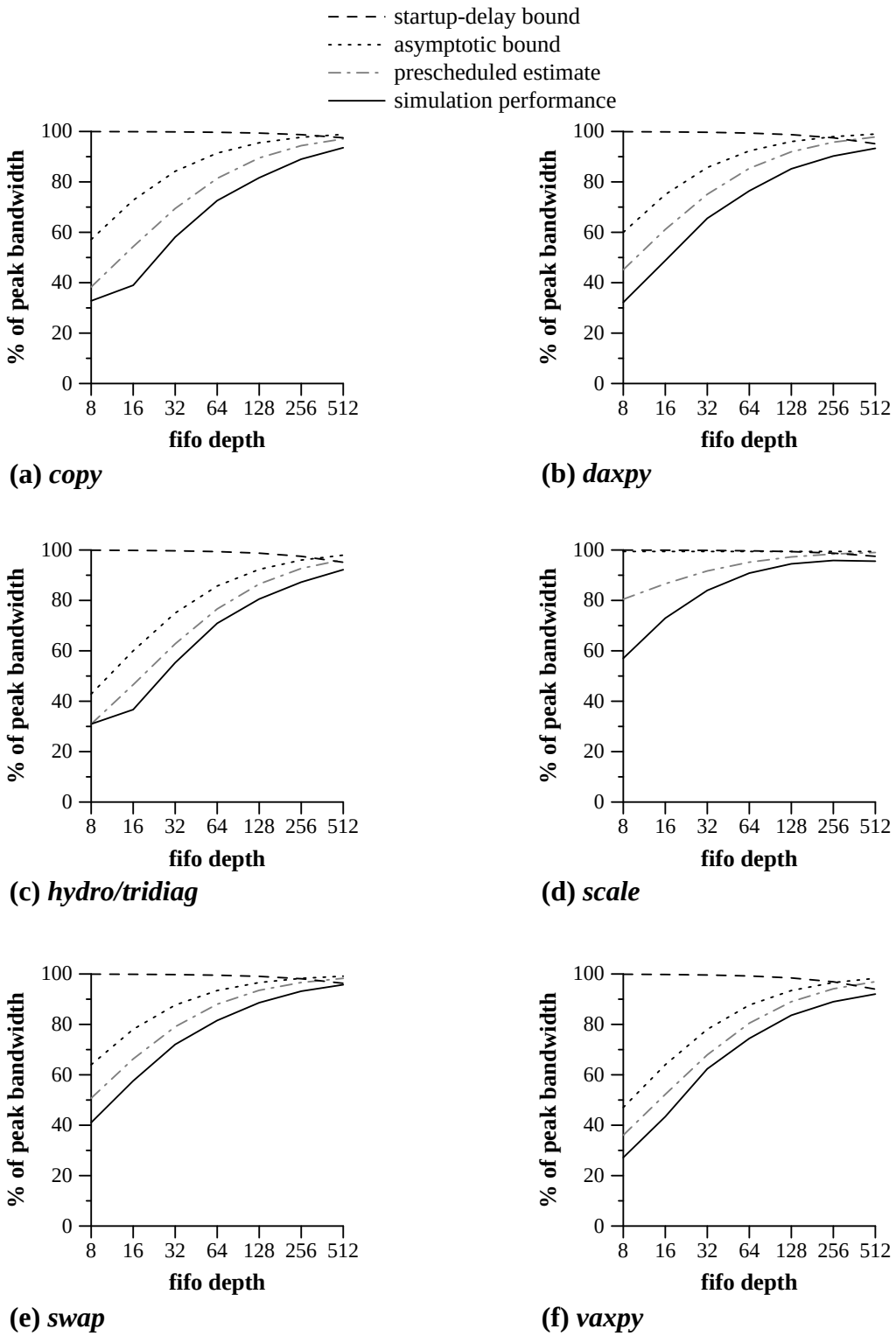
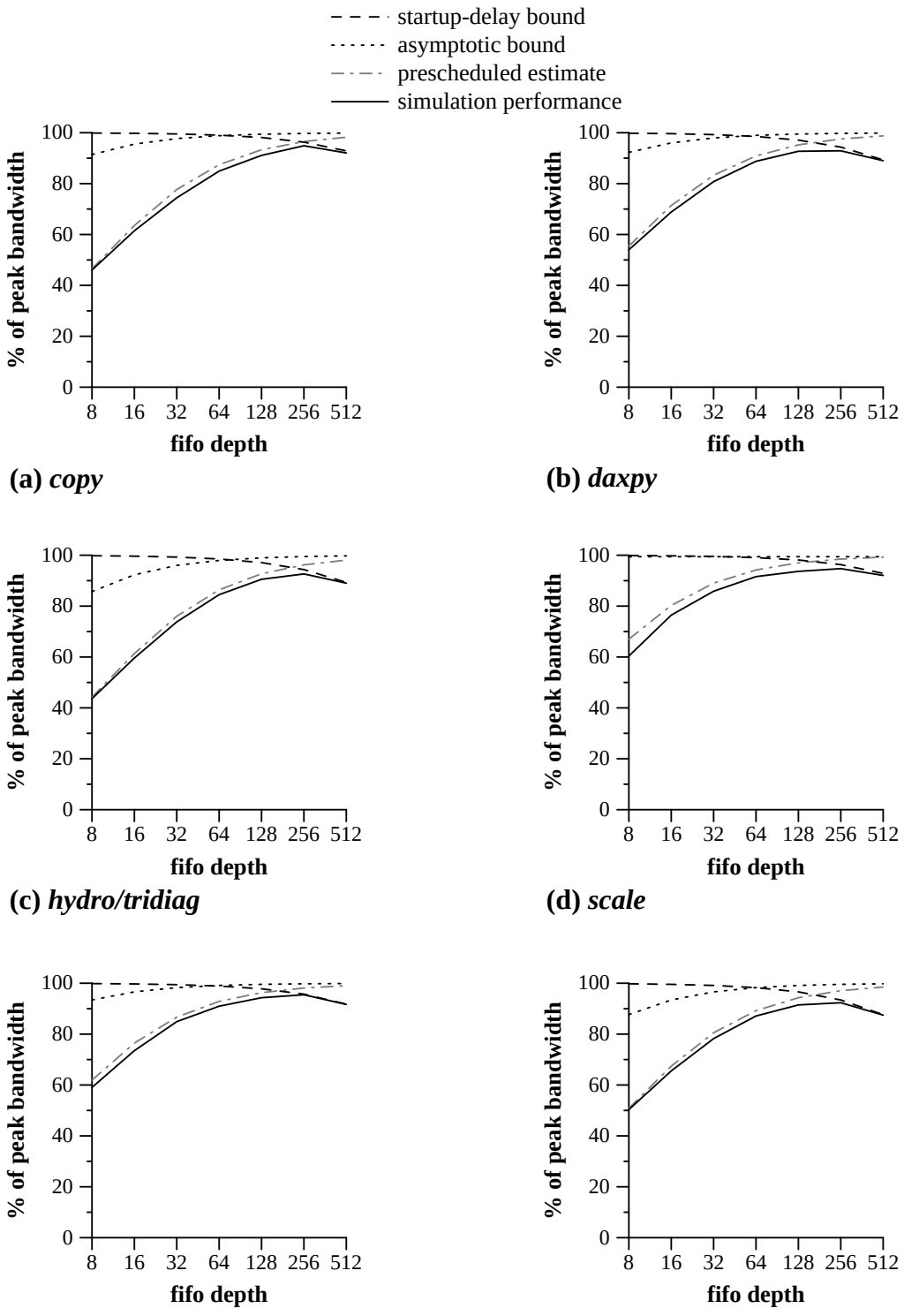


Figure 29 Presched Performance for 2 CEs, 16 Banks, 10,000-Element Vectors



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

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

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

Figure 30 Presched Performance for 4 CEs, 4 Banks, 10,000-Element Vectors

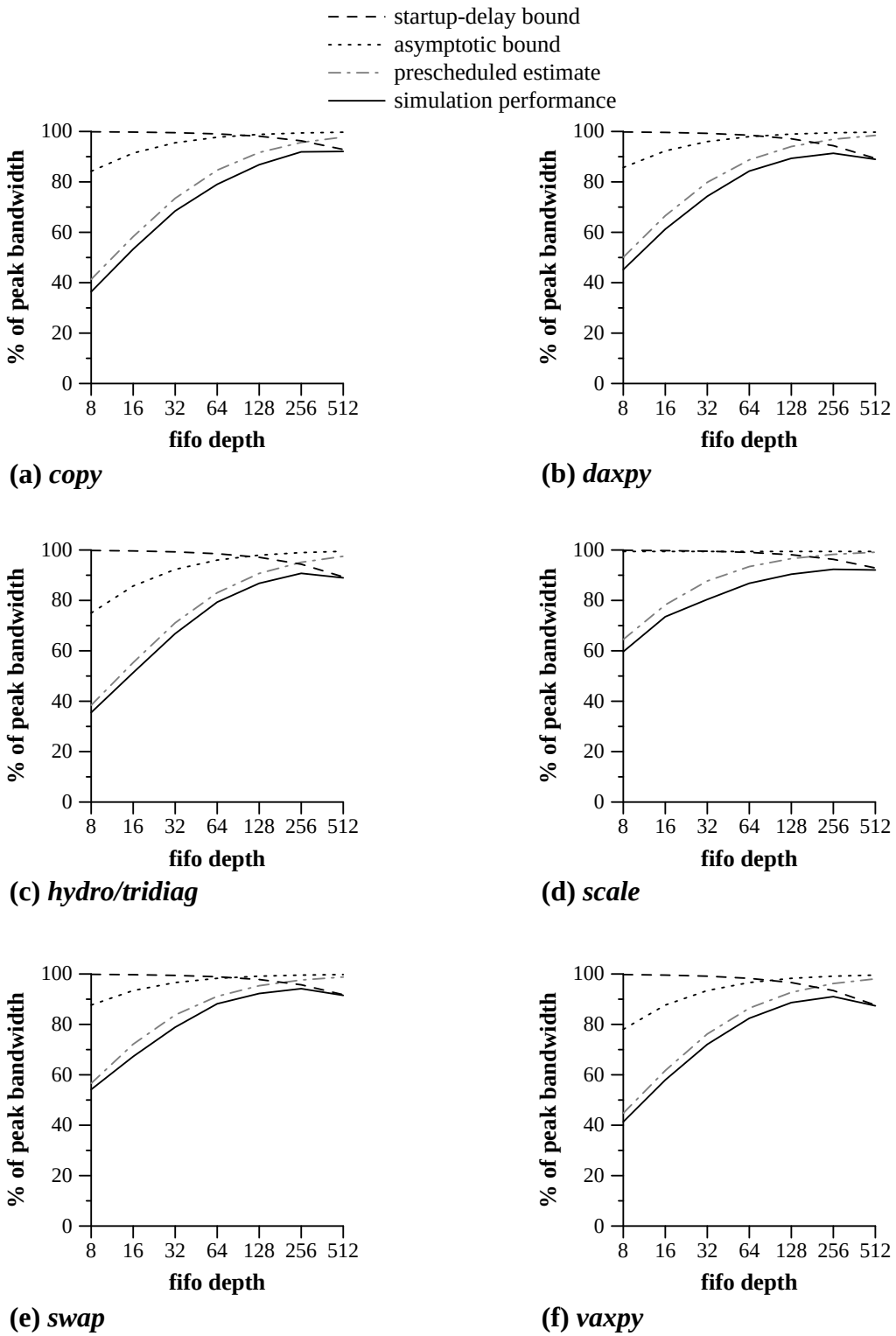


Figure 31 Presched Performance for 4 CEs, 8 Banks, 10,000-Element Vectors

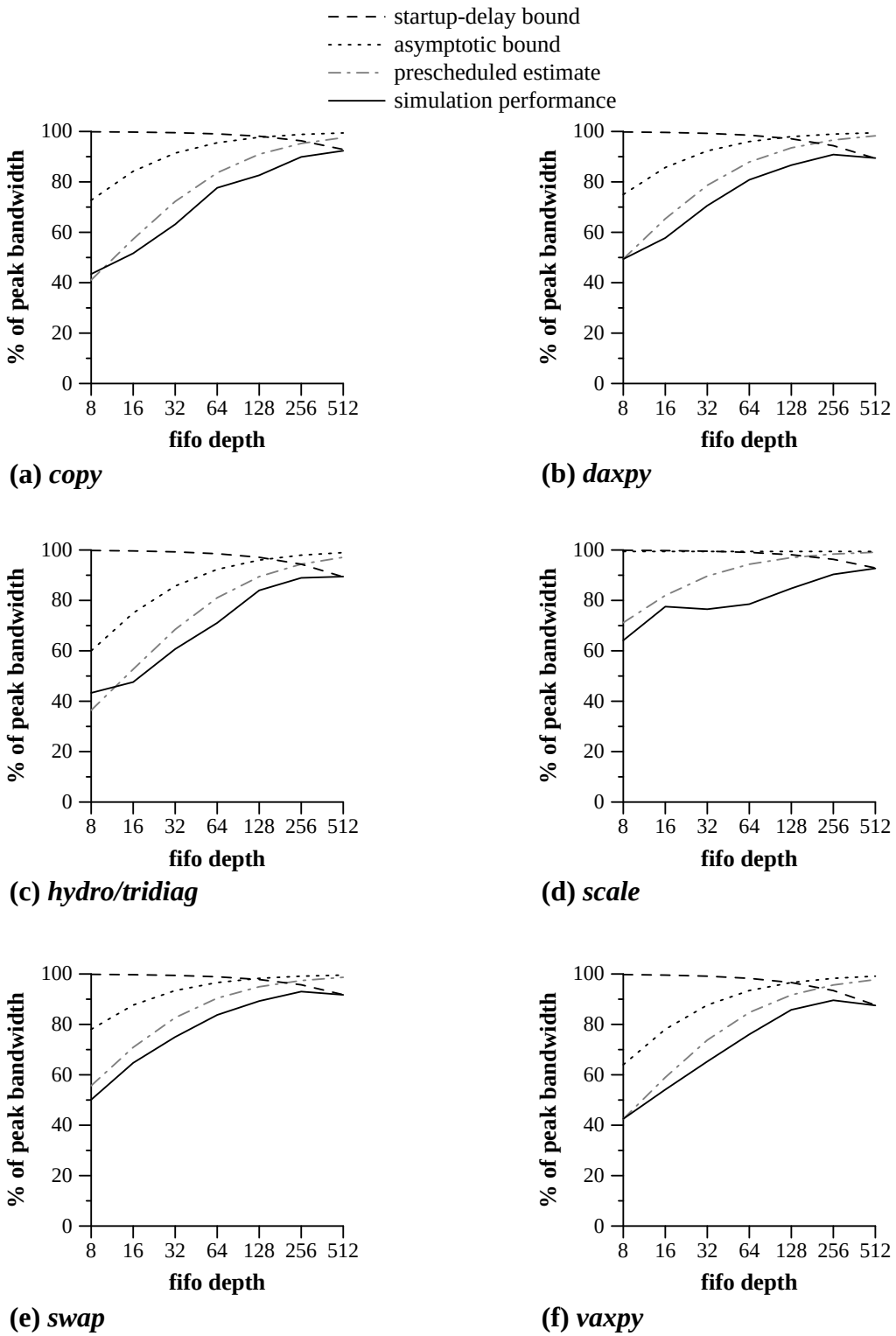


Figure 32 Presched Performance for 4 CEs, 16 Banks, 10,000-Element Vectors

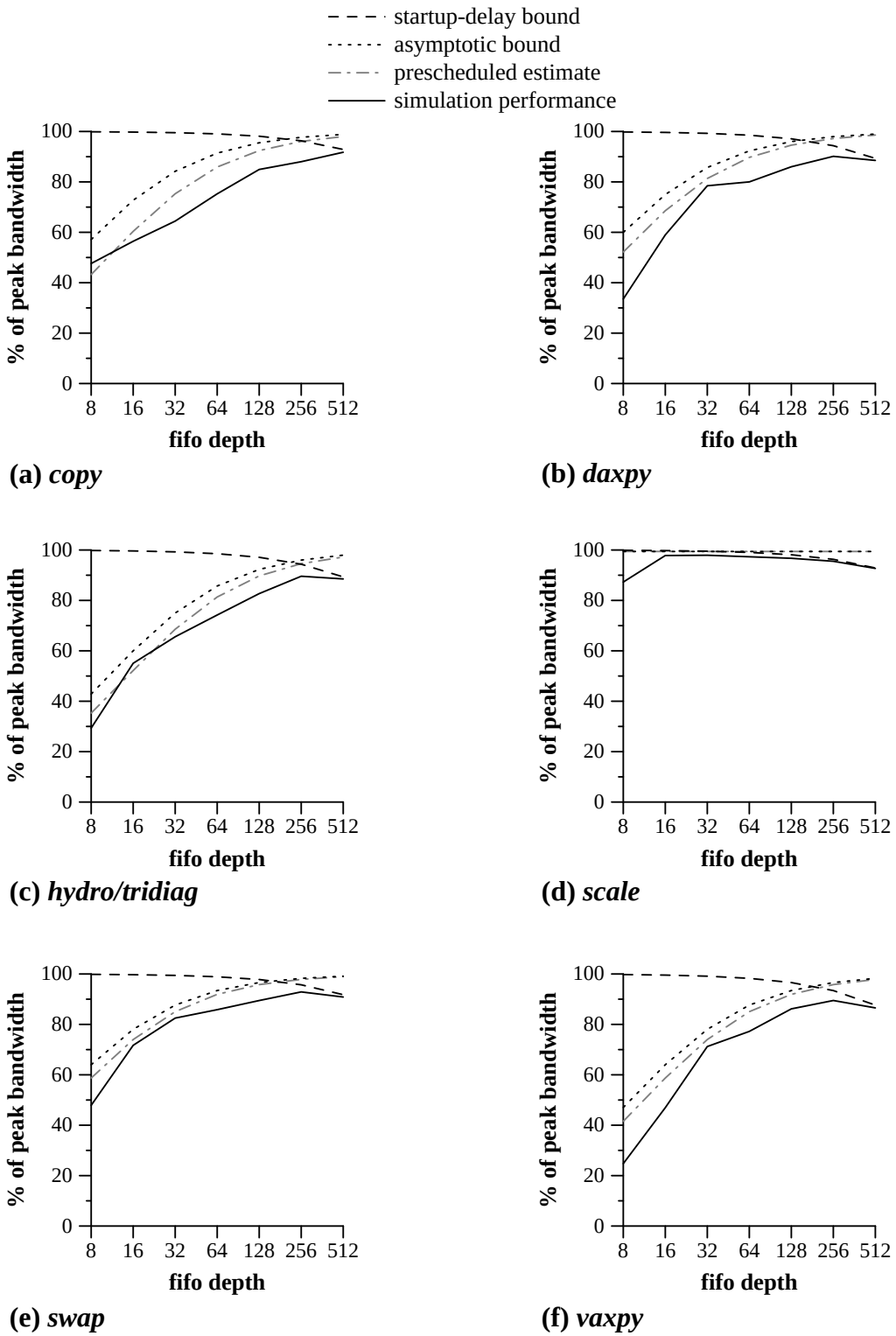


Figure 33 Presched Performance for 4 CEs, 32 Banks, 10,000-Element Vectors

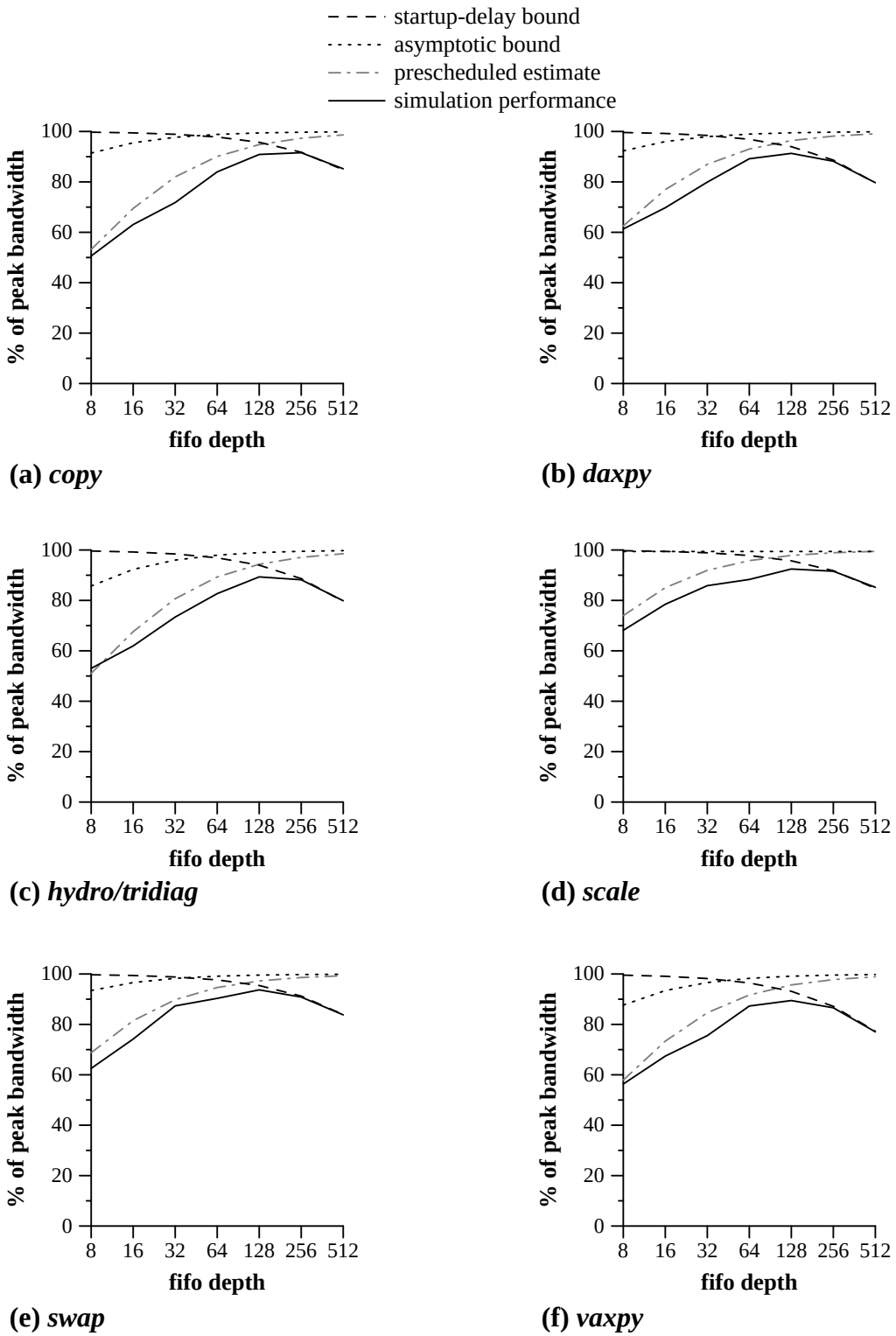


Figure 34 Presched Performance for 8 CEs, 8 Banks, 10,000-Element Vectors

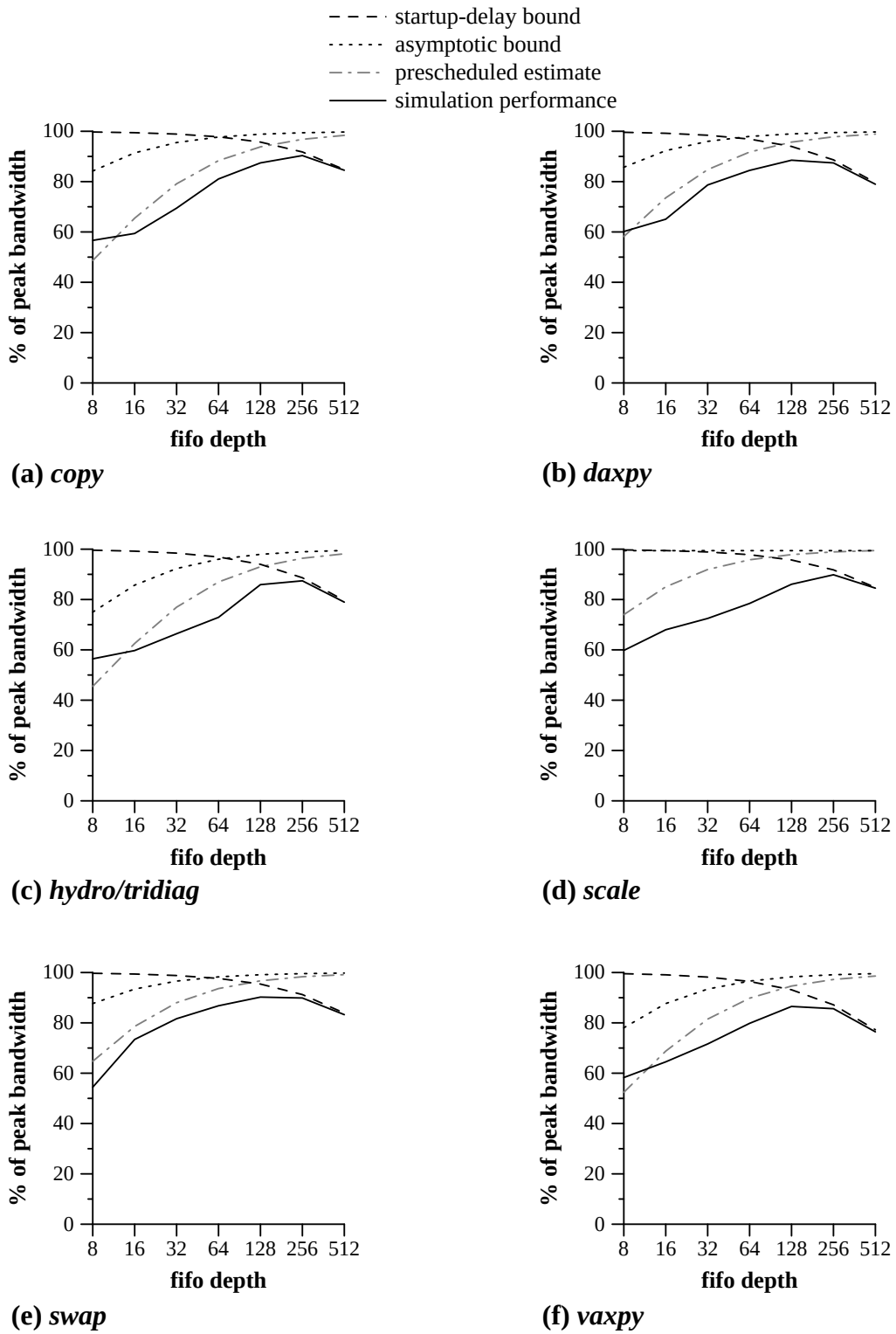


Figure 35 Presched Performance for 8 CEs, 16 Banks, 10,000-Element Vectors

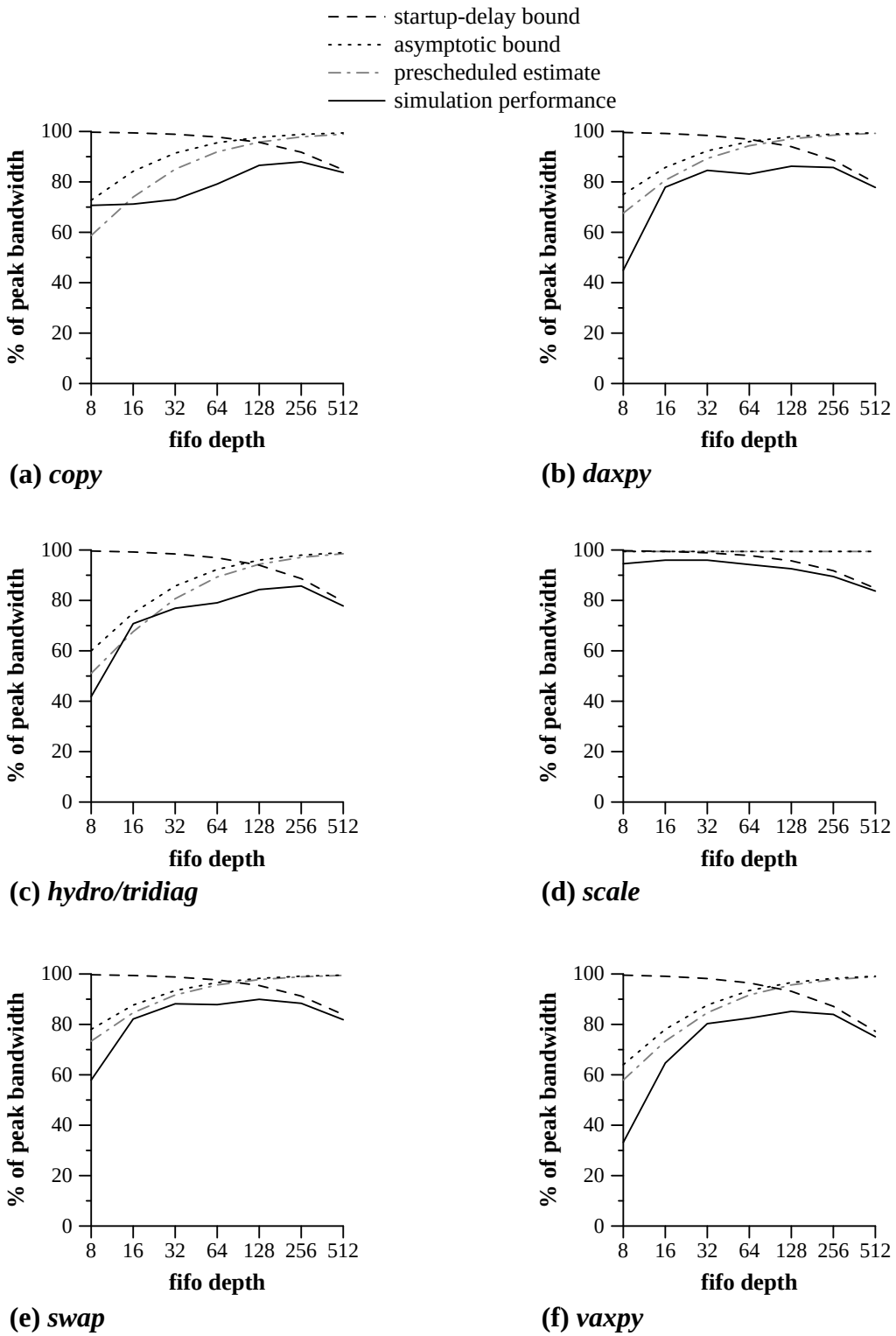


Figure 36 Presched Performance for 8 CEs, 32 Banks, 10,000-Element Vectors

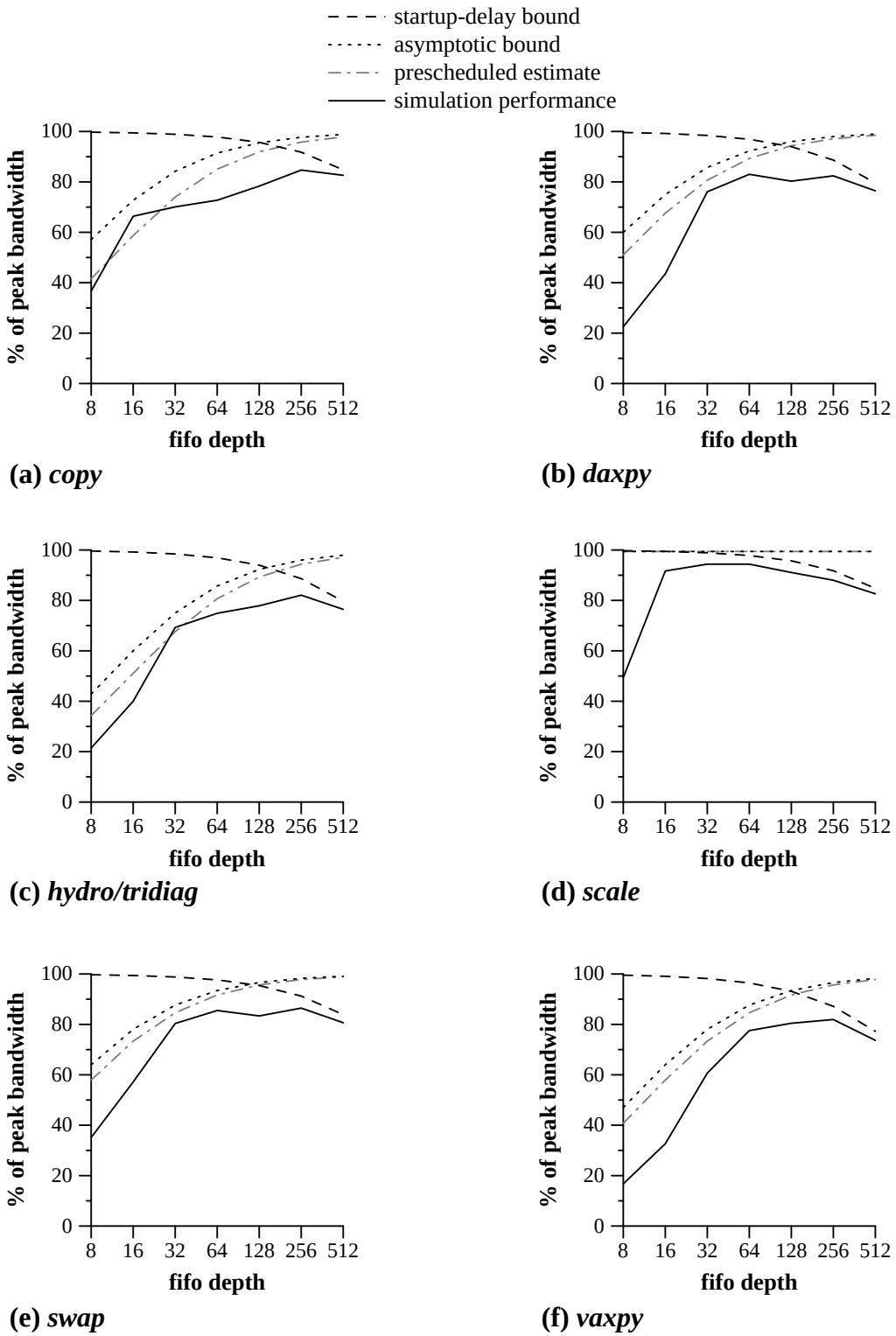


Figure 37 Presched Performance for 8 CEs, 64 Banks, 10,000-Element Vectors

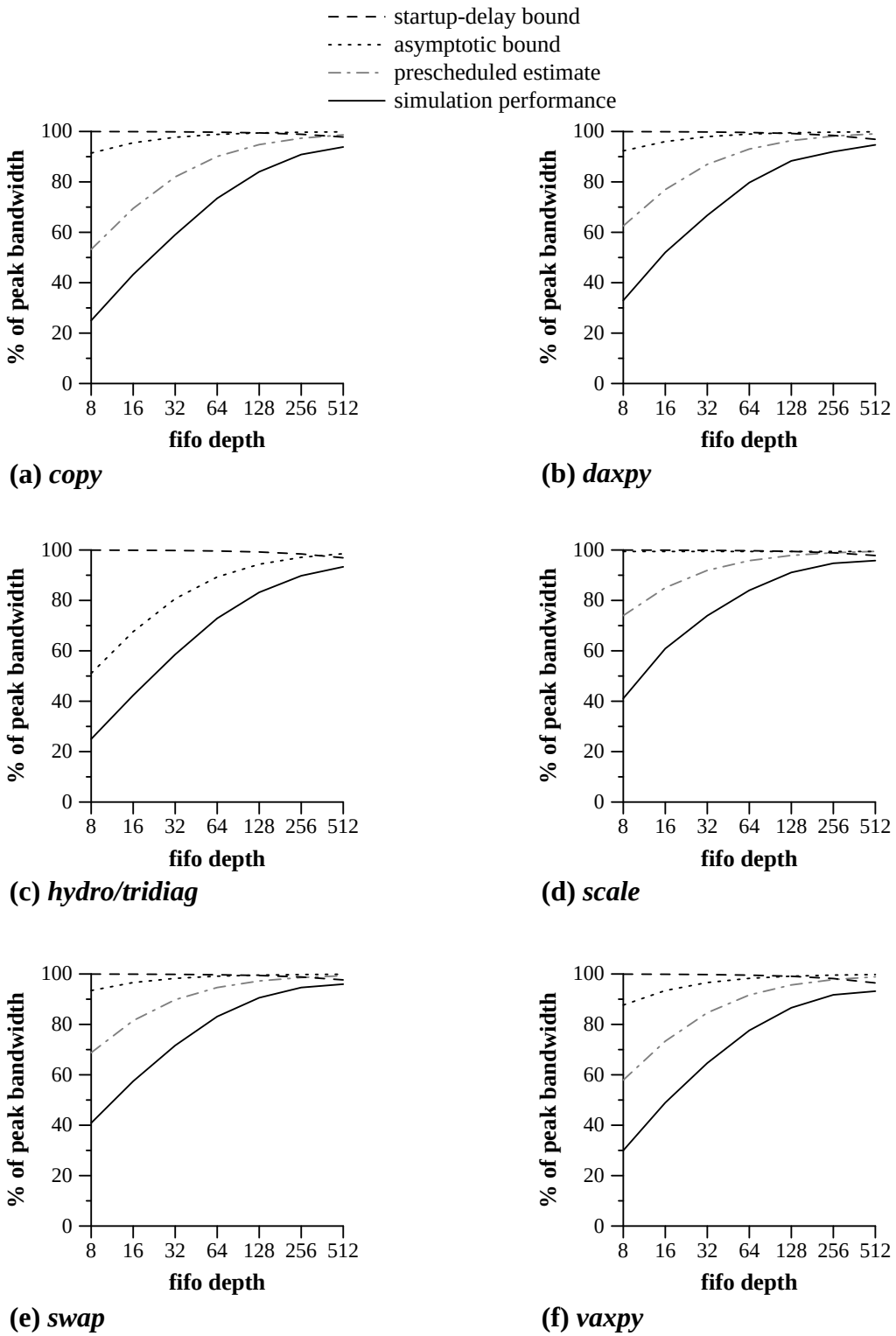


Figure 38 Presched Performance for 8 CEs, 8 Banks, 80,000-Element Vectors

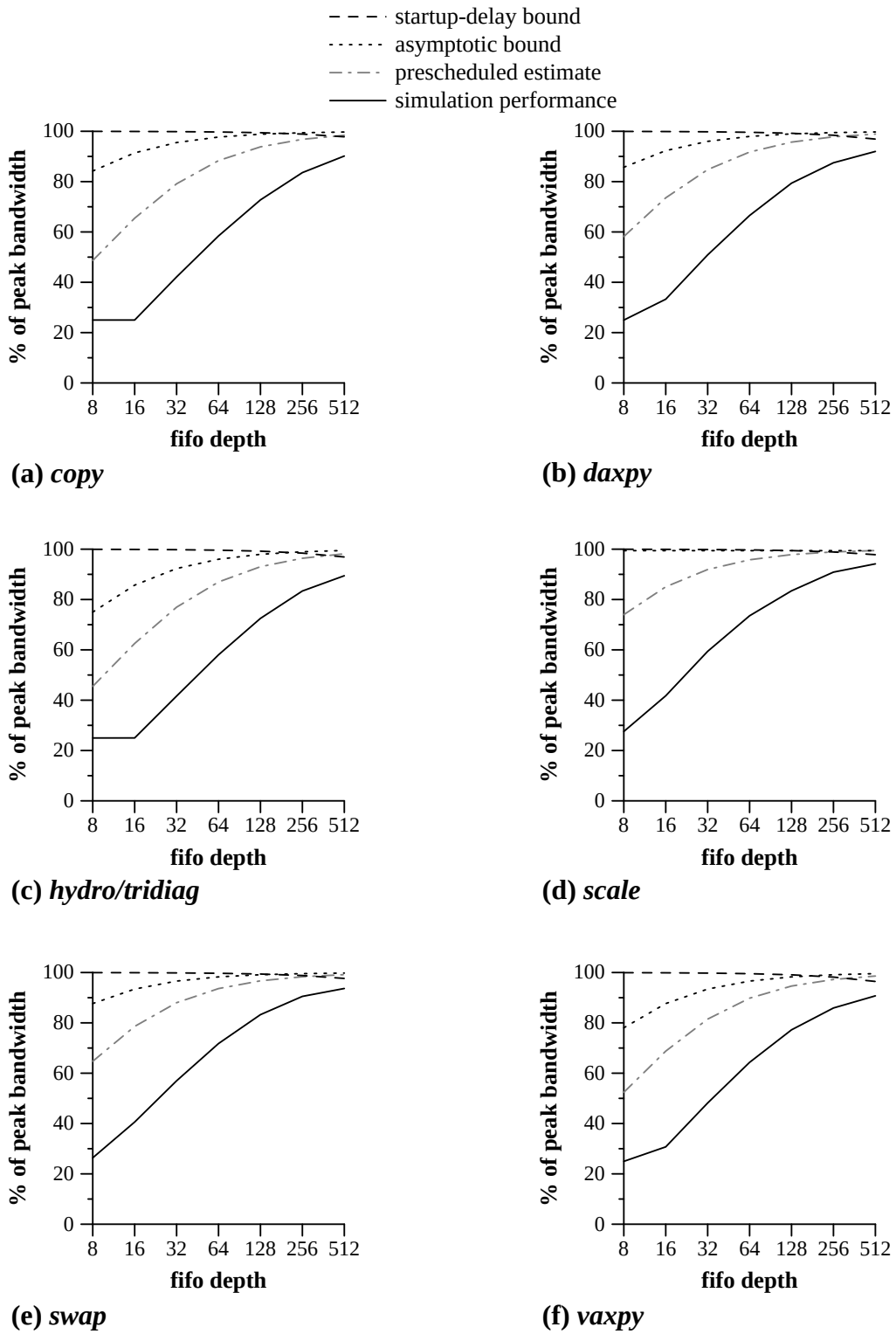


Figure 39 Presched Performance for 8 CEs, 16 Banks, 80,000-Element Vectors

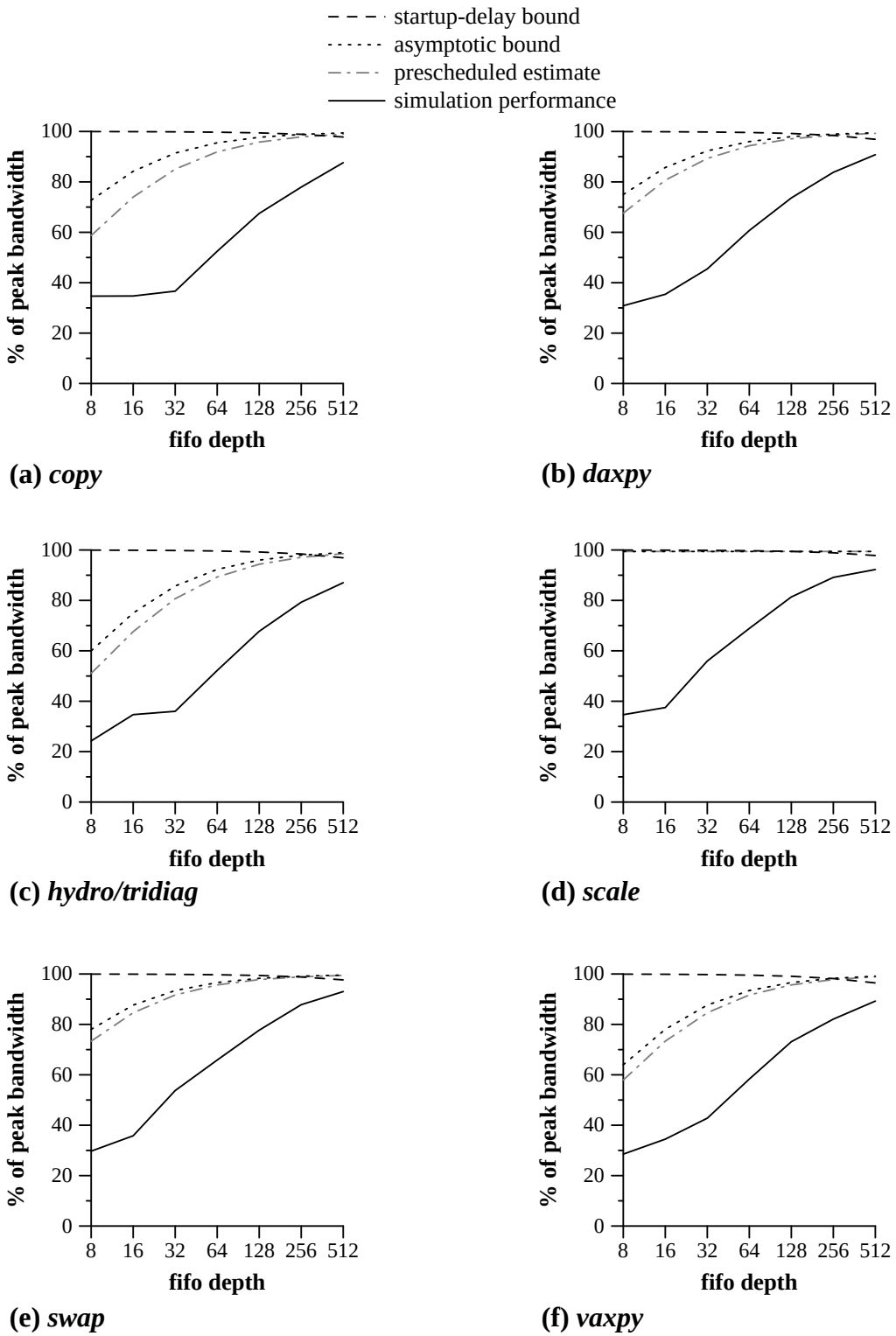


Figure 40 Presched Performance for 8 CEs, 32 Banks, 80,000-Element Vectors

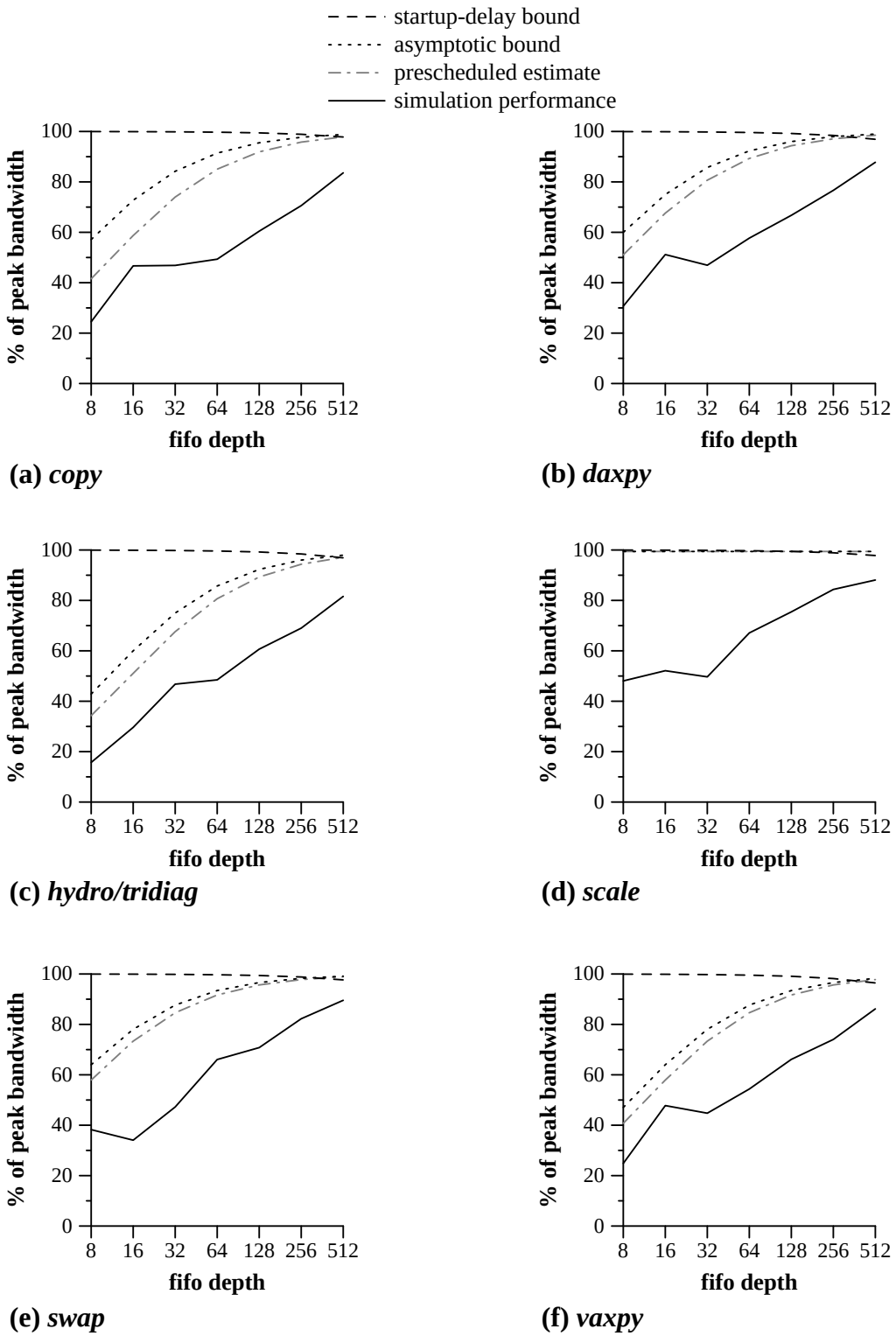


Figure 41 Presched Performance for 8 CEs, 64 Banks, 80,000-Element Vectors

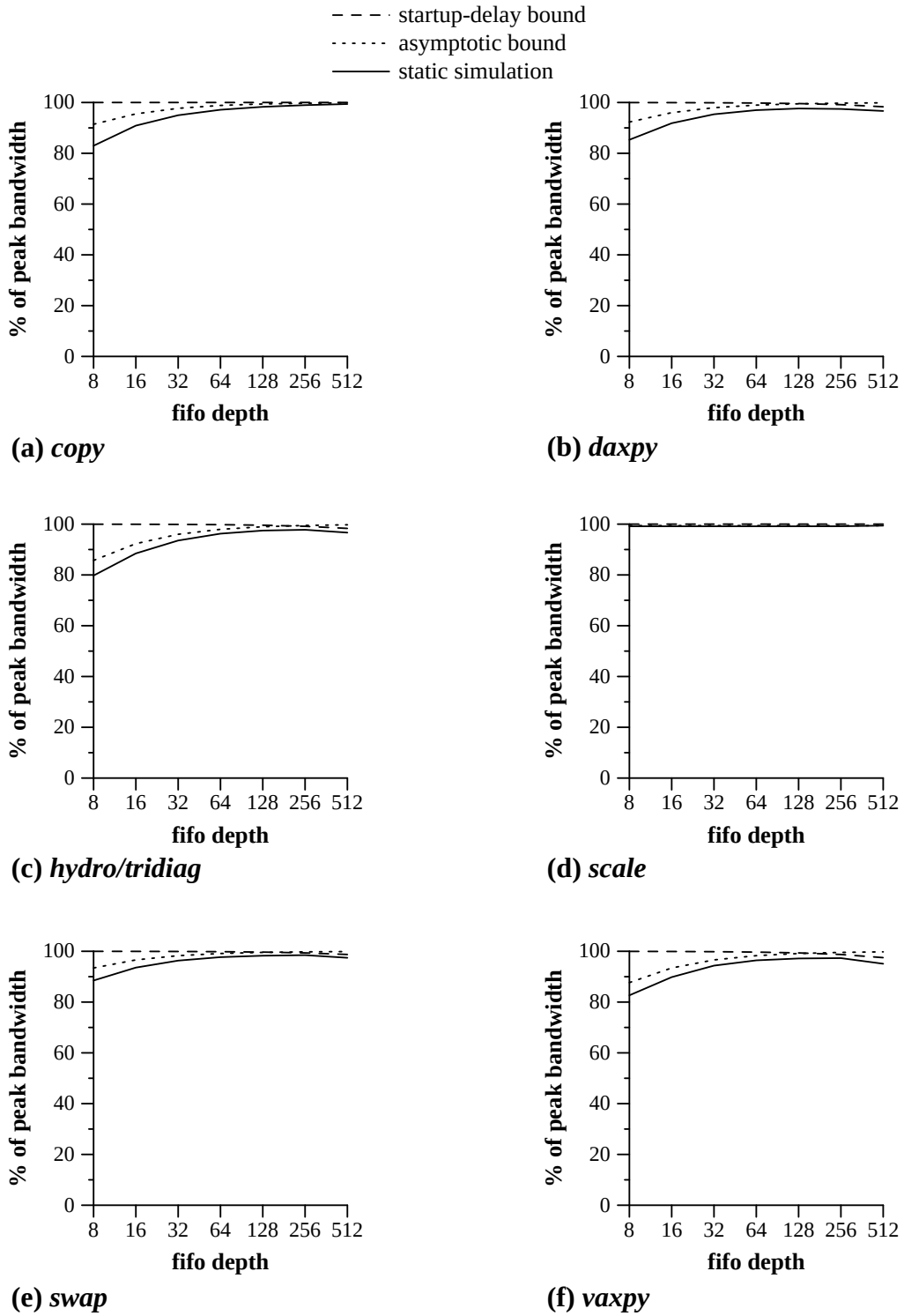


Figure 42 Static Performance for 2 CEs, 2 Banks, 10,000-Element Vectors

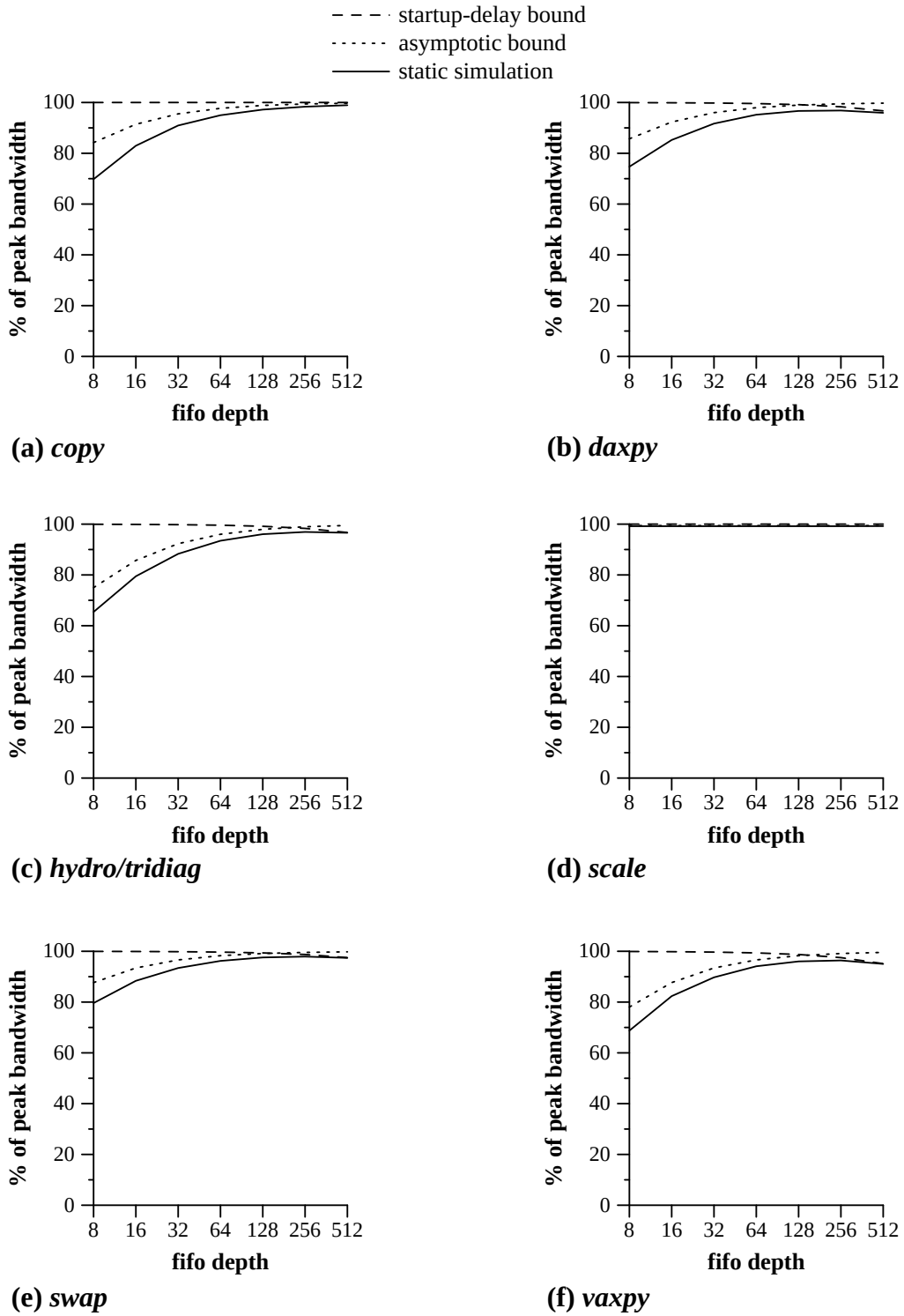


Figure 43 Static Performance for 2 CEs, 4 Banks, 10,000-Element Vectors

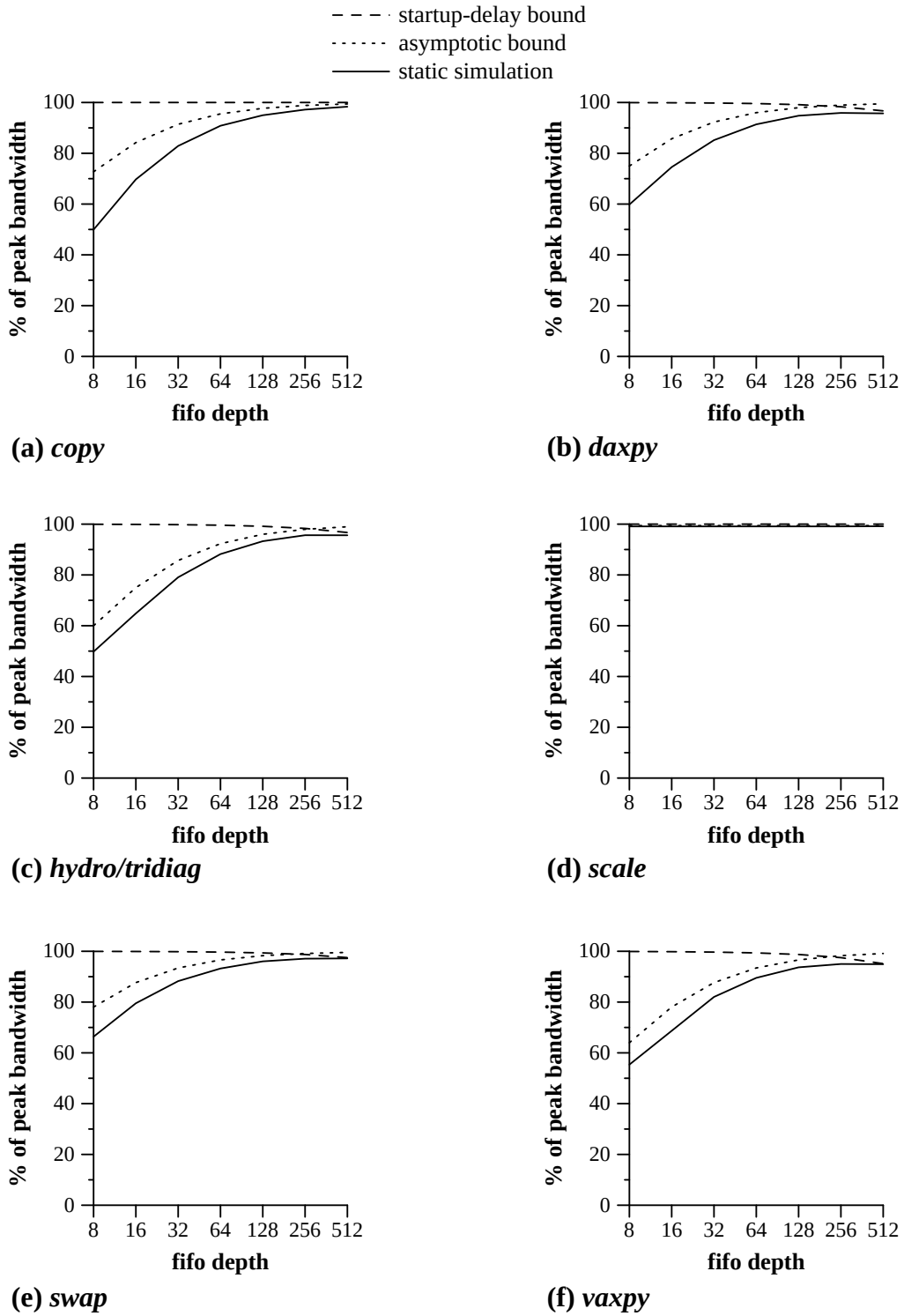


Figure 44 Static Performance for 2 CEs, 8 Banks, 10,000-Element Vectors

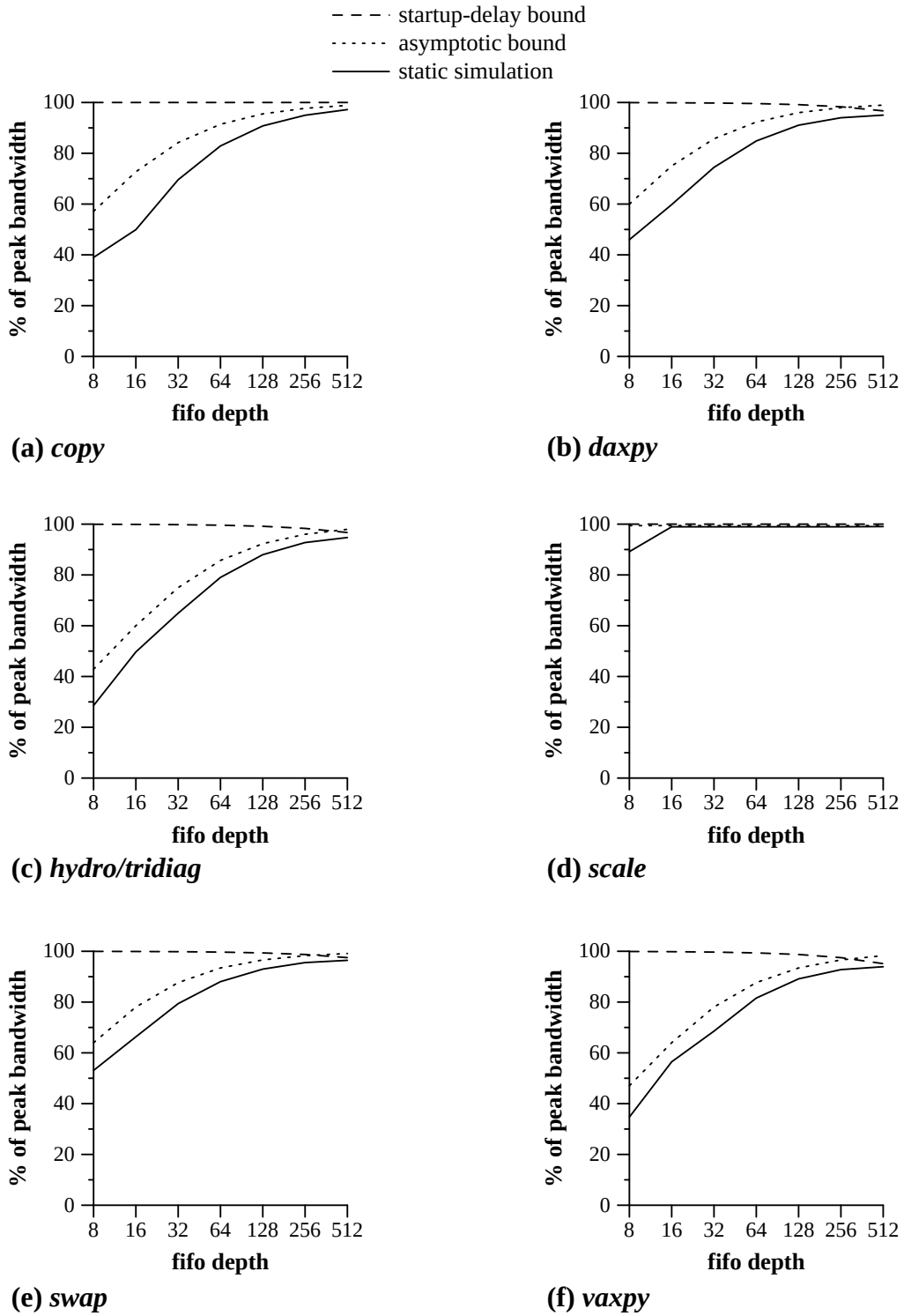


Figure 45 Static Performance for 2 CEs, 16 Banks, 10,000-Element Vectors

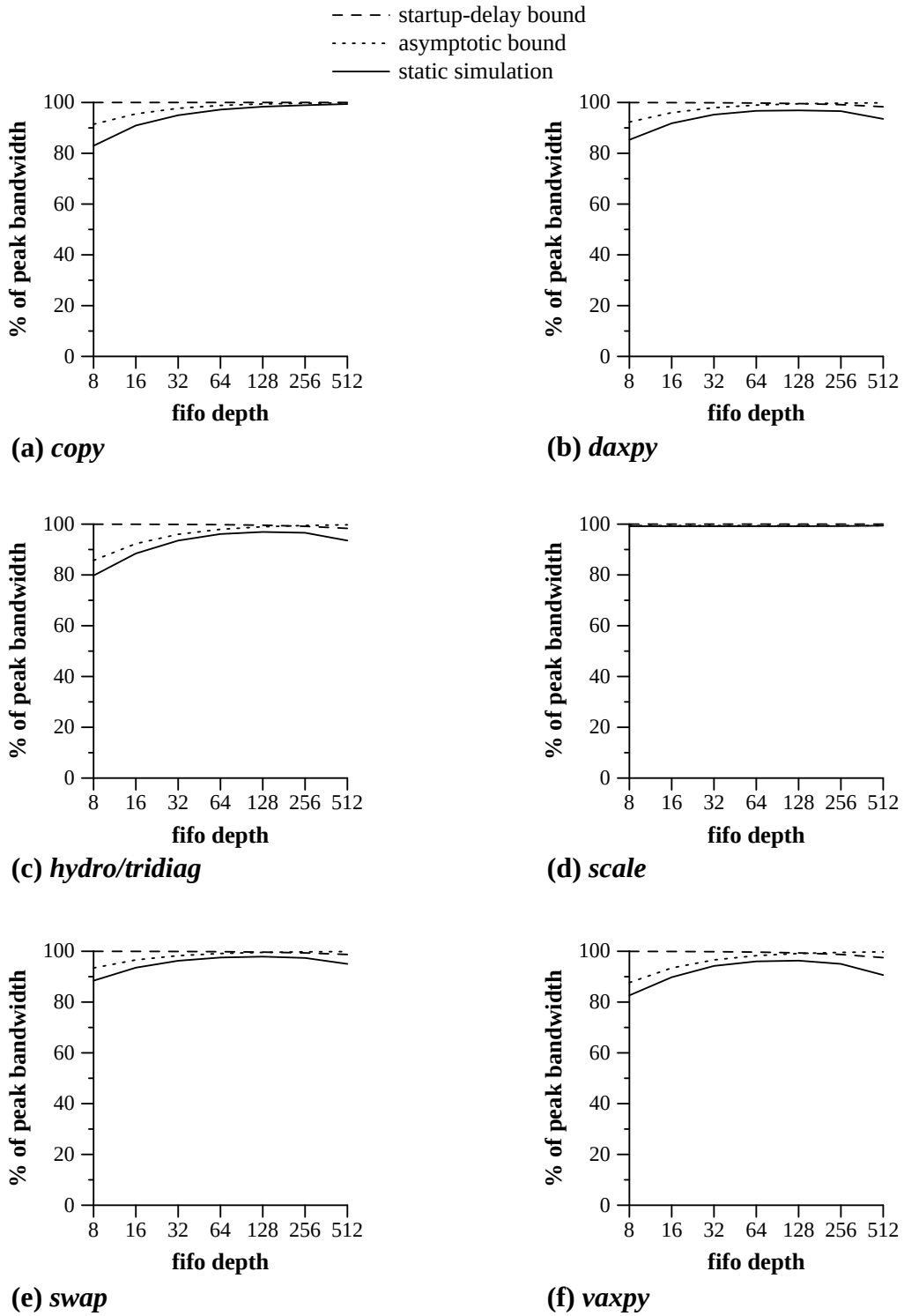
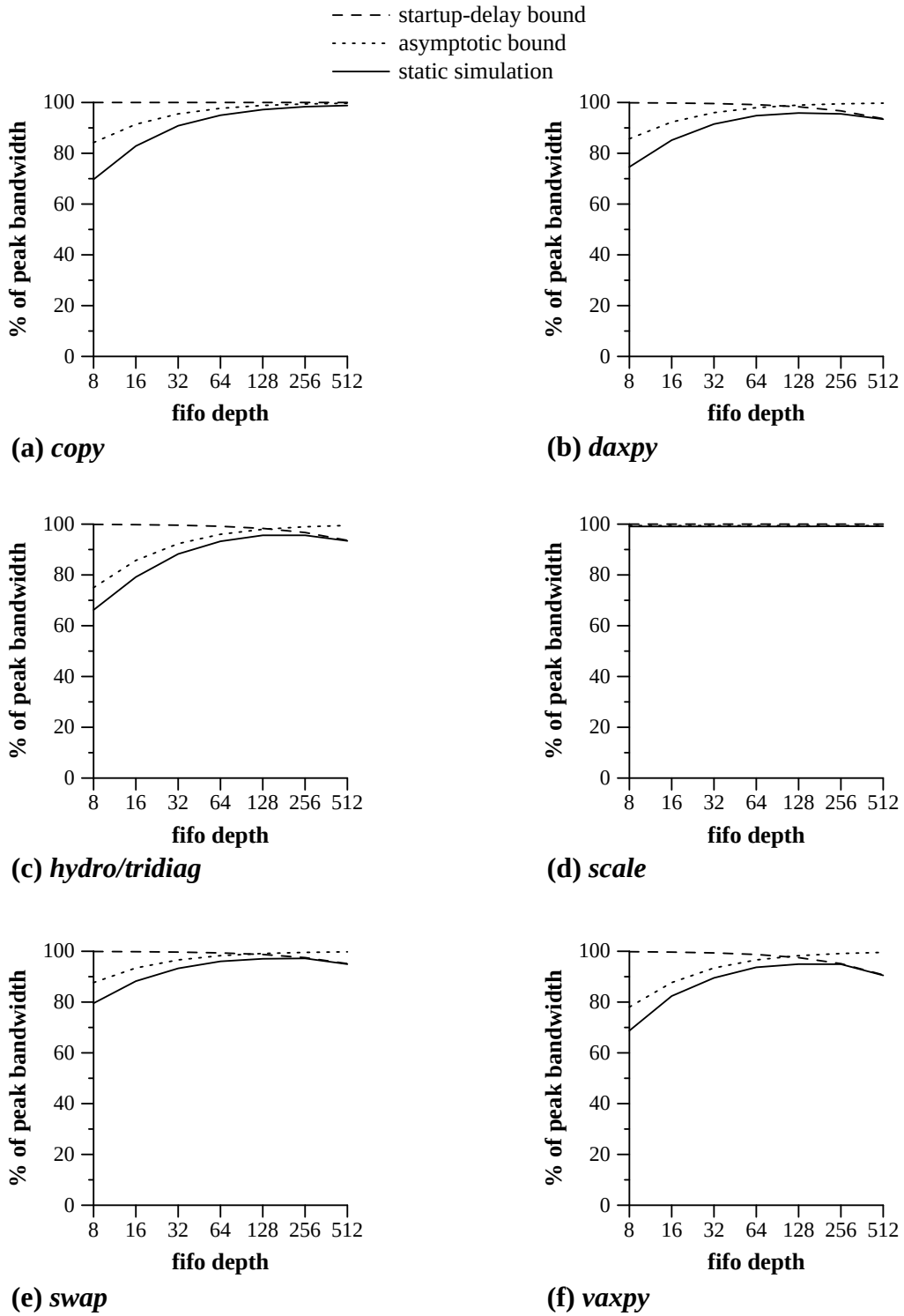


Figure 46 Static Performance for 4 CEs, 4 Banks, 10,000-Element Vectors



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

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

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

Figure 47 Static Performance for 4 CEs, 8 Banks, 10,000-Element Vectors

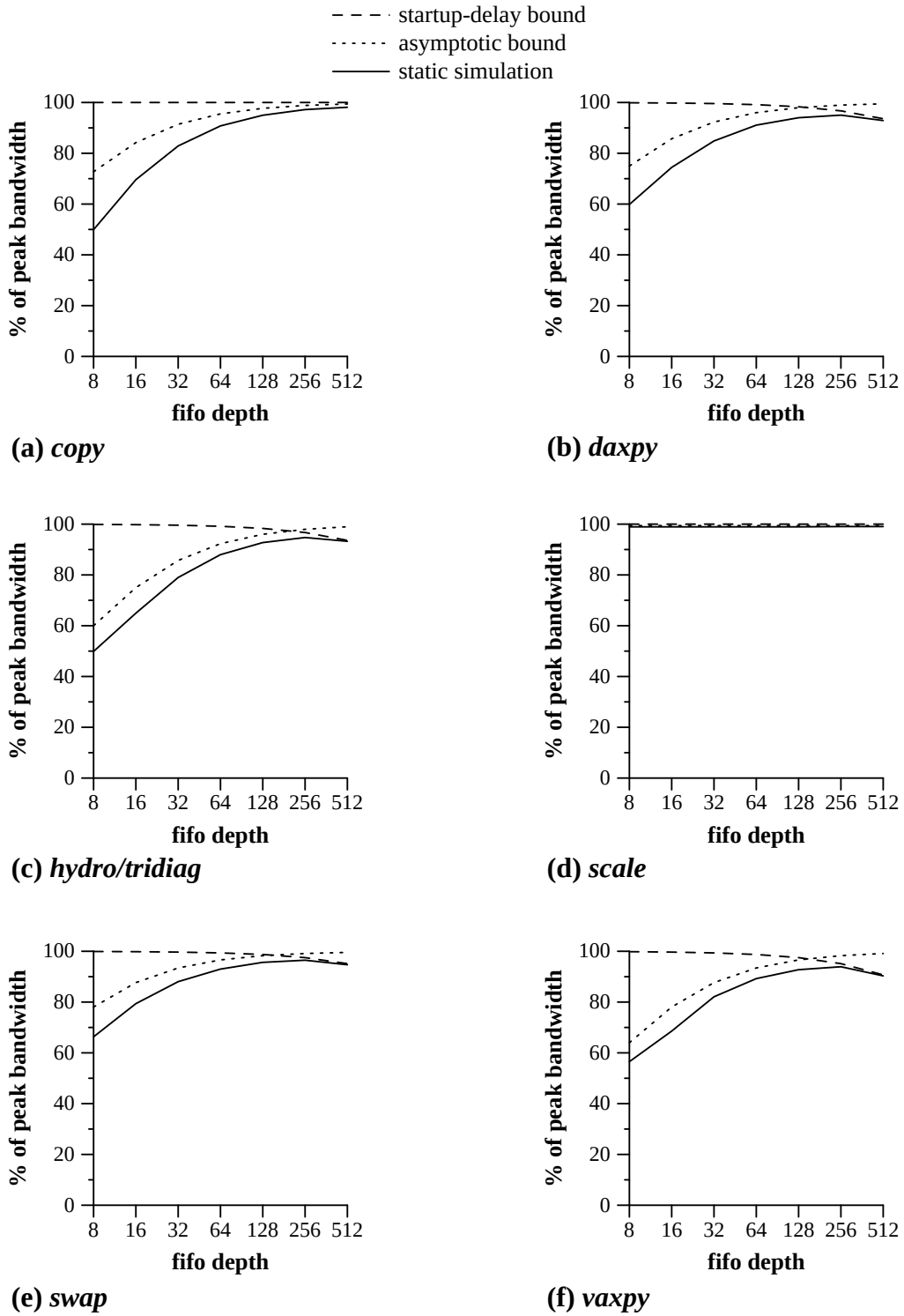


Figure 48 Static Performance for 4 CEs, 16 Banks, 10,000-Element Vectors

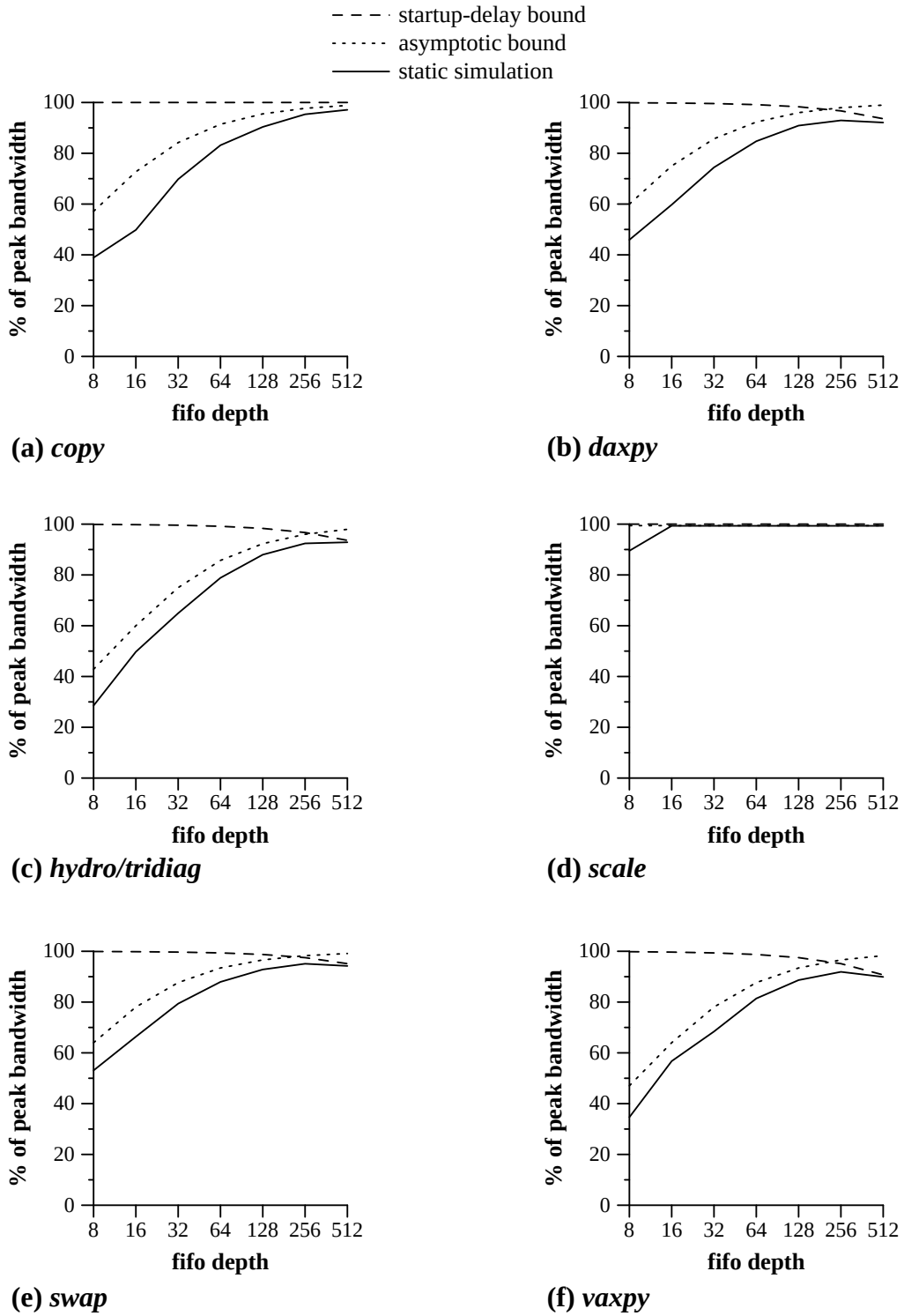


Figure 49 Static Performance for 4 CEs, 32 Banks, 10,000-Element Vectors

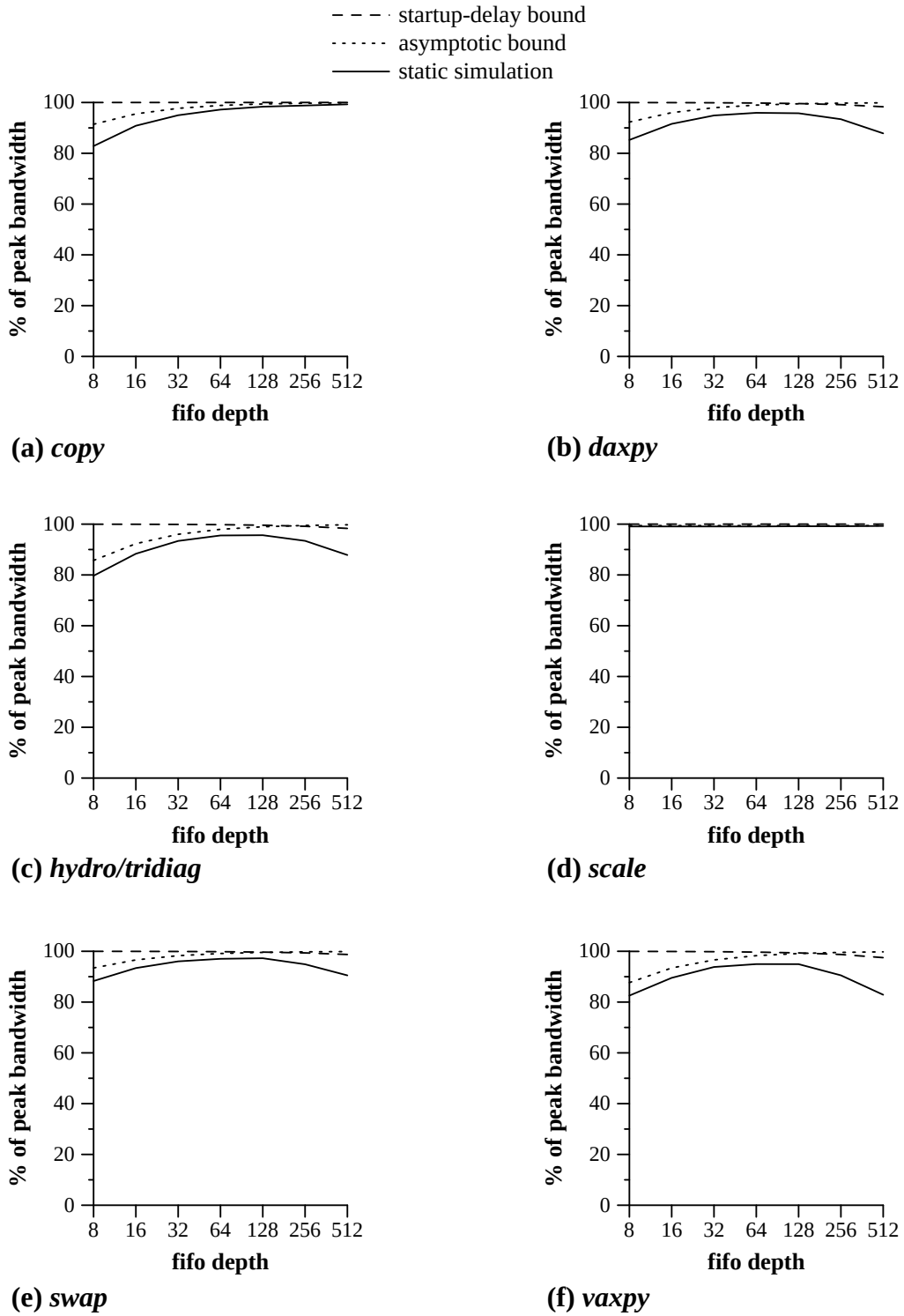


Figure 50 Static Performance for 8 CEs, 8 Banks, 10,000-Element Vectors

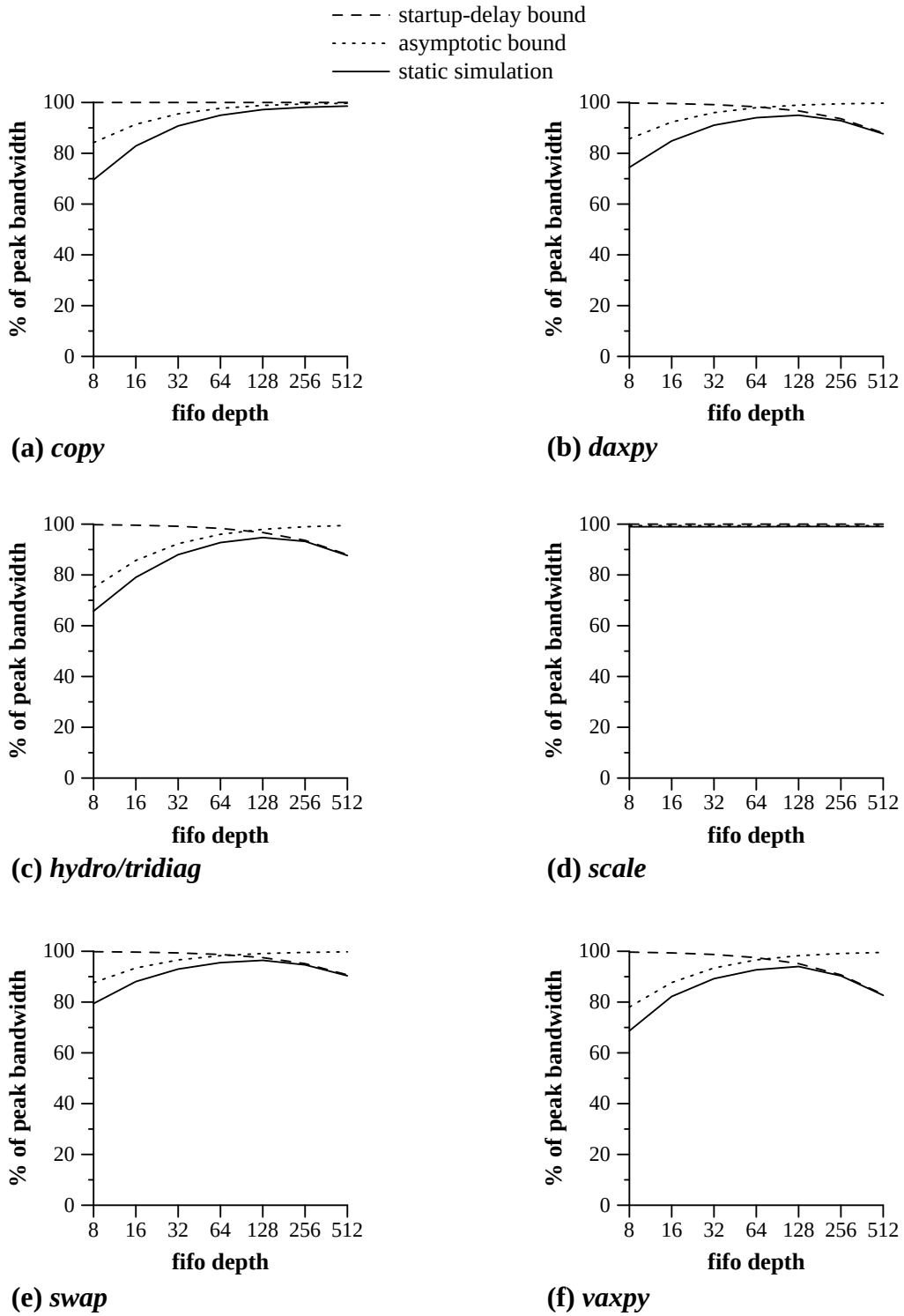
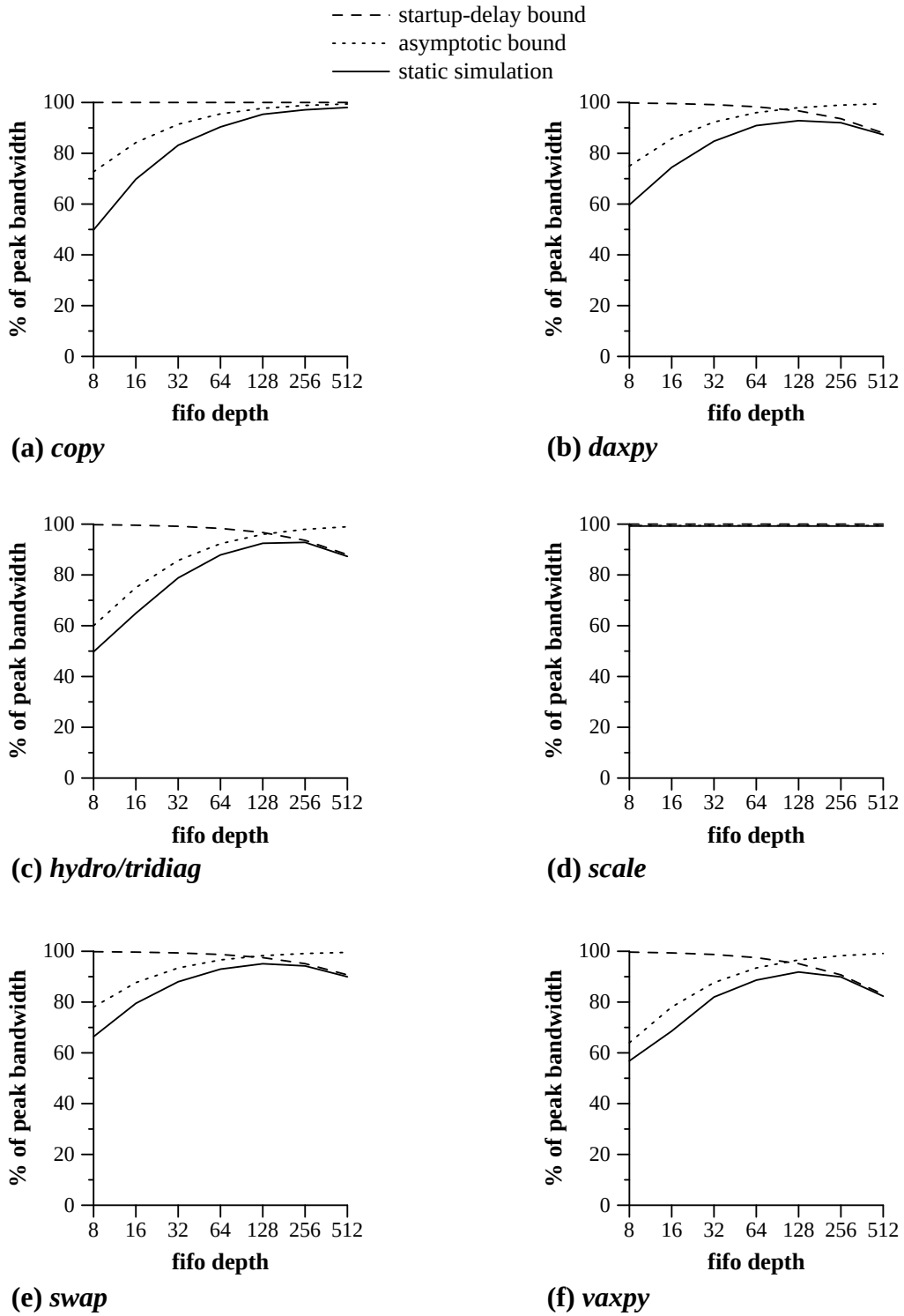


Figure 51 Static Performance for 8 CEs, 16 Banks, 10,000-Element Vectors

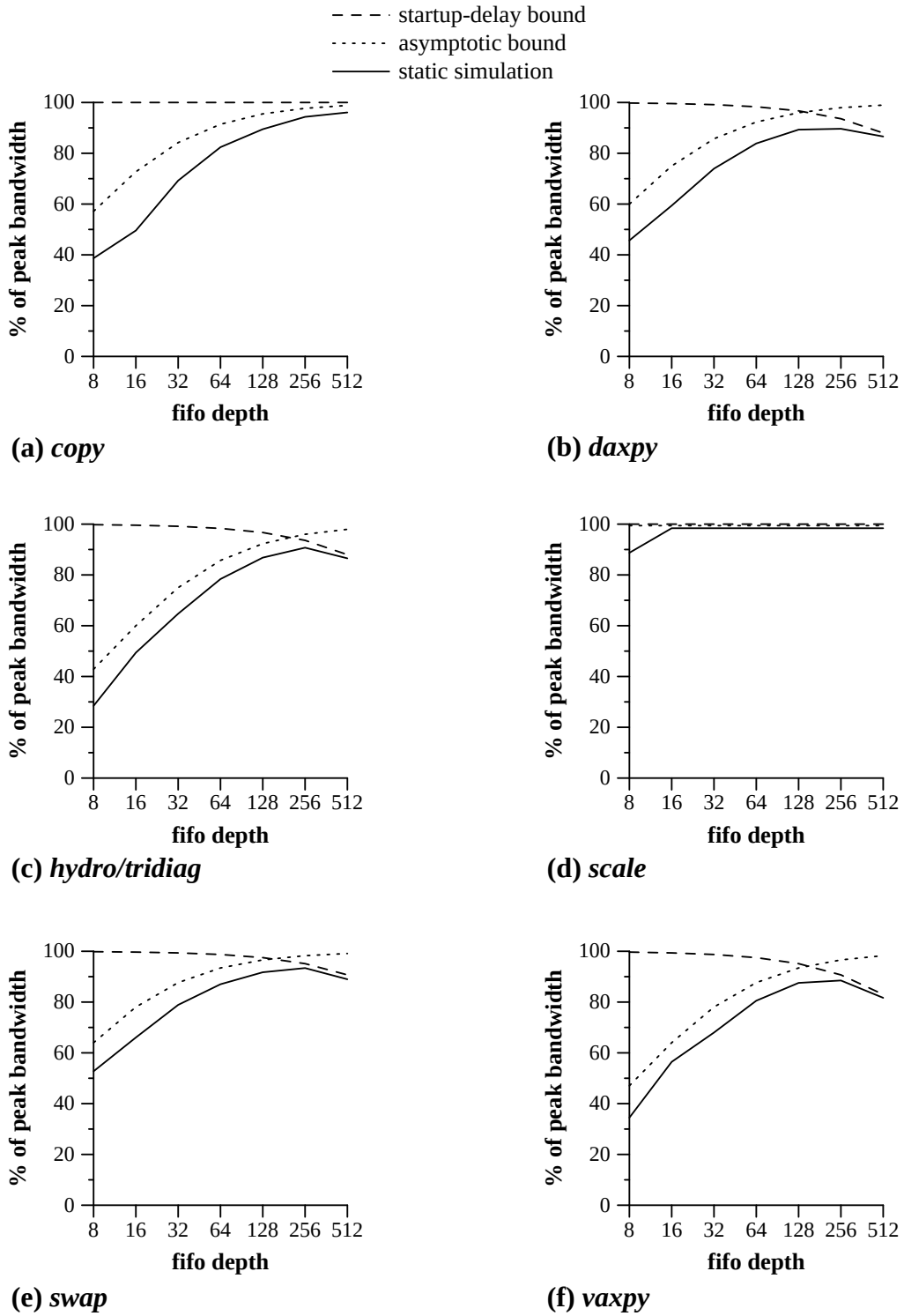


(a) ***copy*** **(b)** ***daxpy***

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

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

Figure 52 Static Performance for 8 CEs, 32 Banks, 10,000-Element Vectors



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

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

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

Figure 53 Static Performance for 8 CEs, 64 Banks, 10,000-Element Vectors

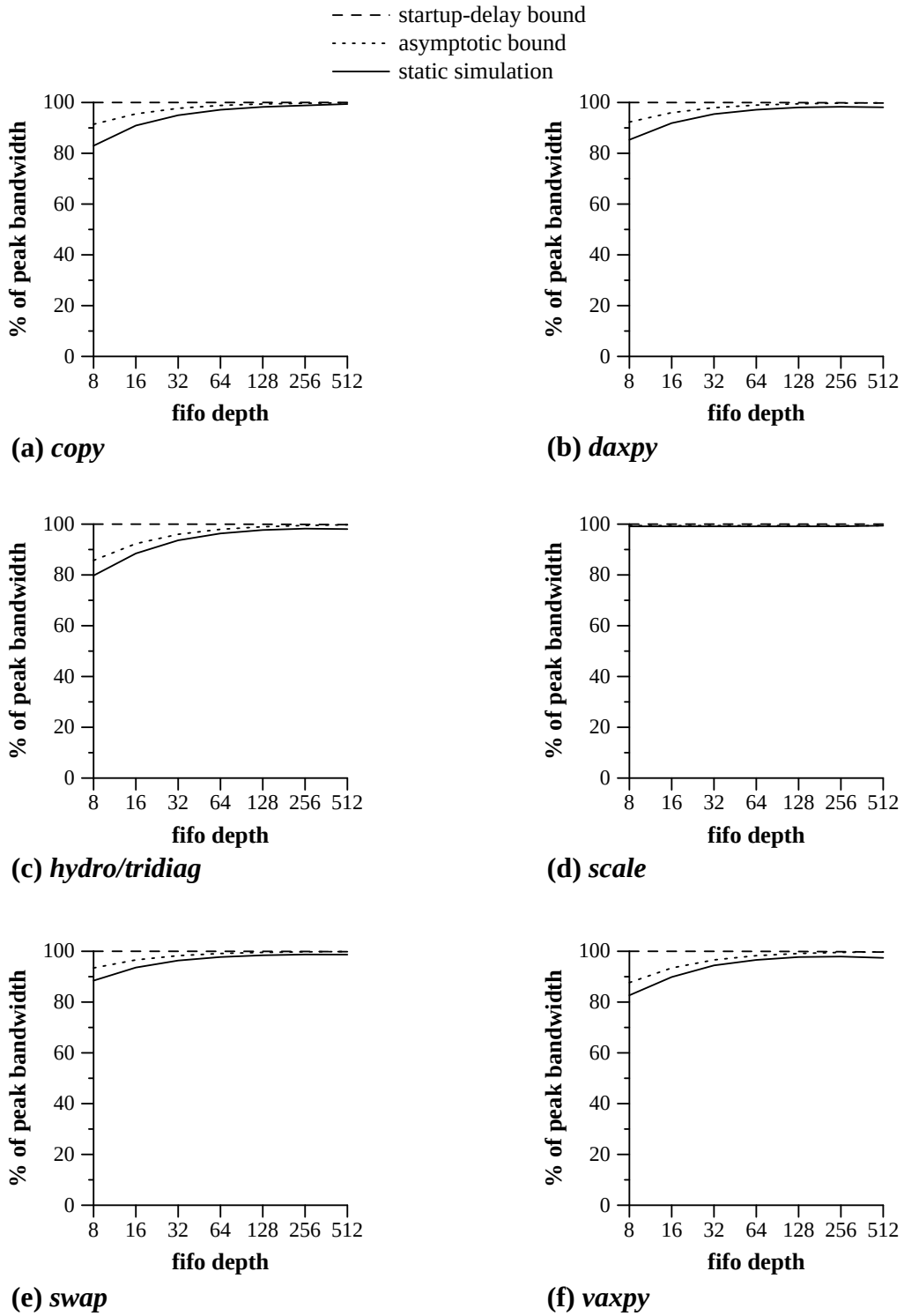


Figure 54 Static Performance for 8 CEs, 8 Banks, 80,000-Element Vectors

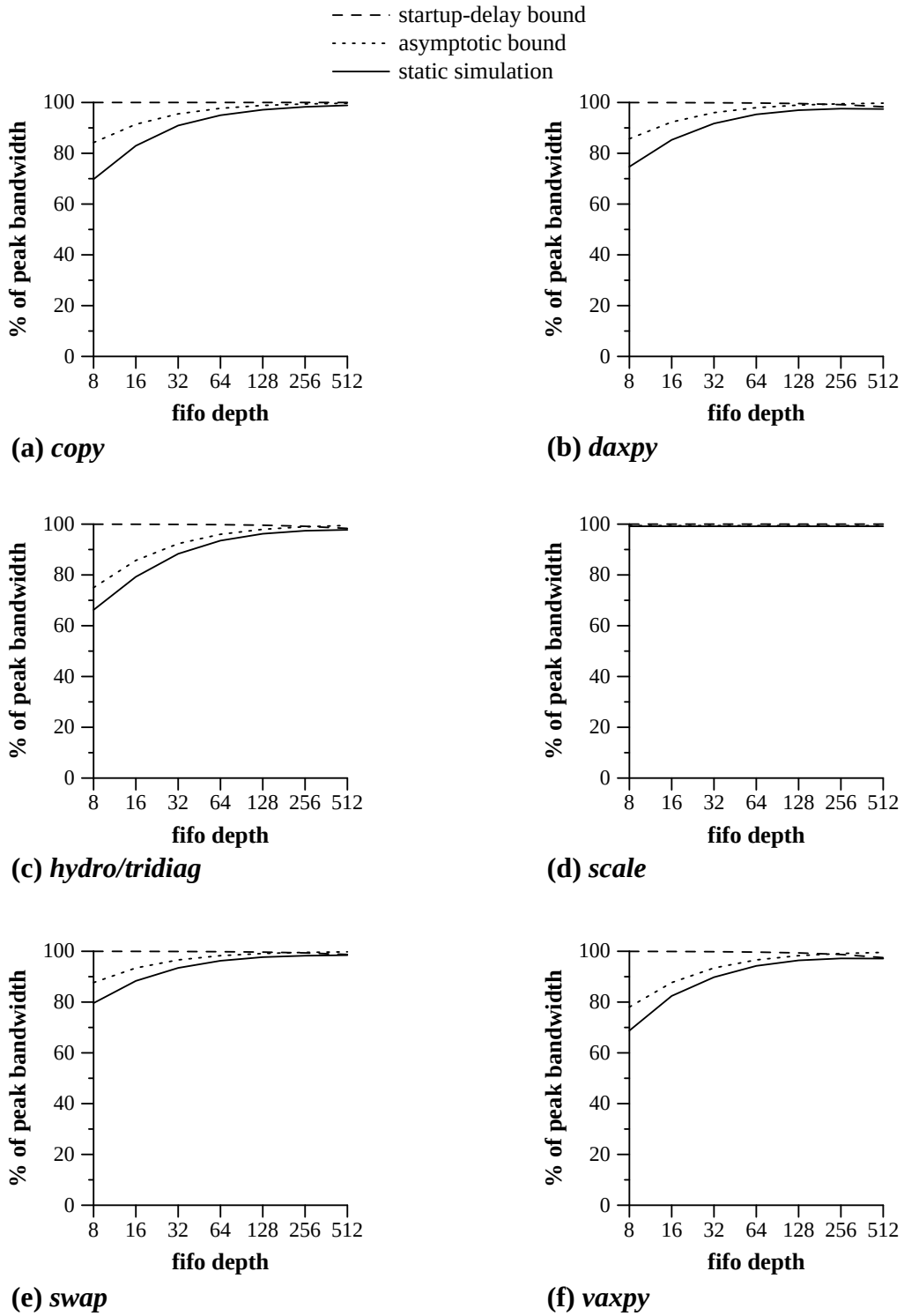


Figure 55 Static Performance for 8 CEs, 16 Banks, 80,000-Element Vectors

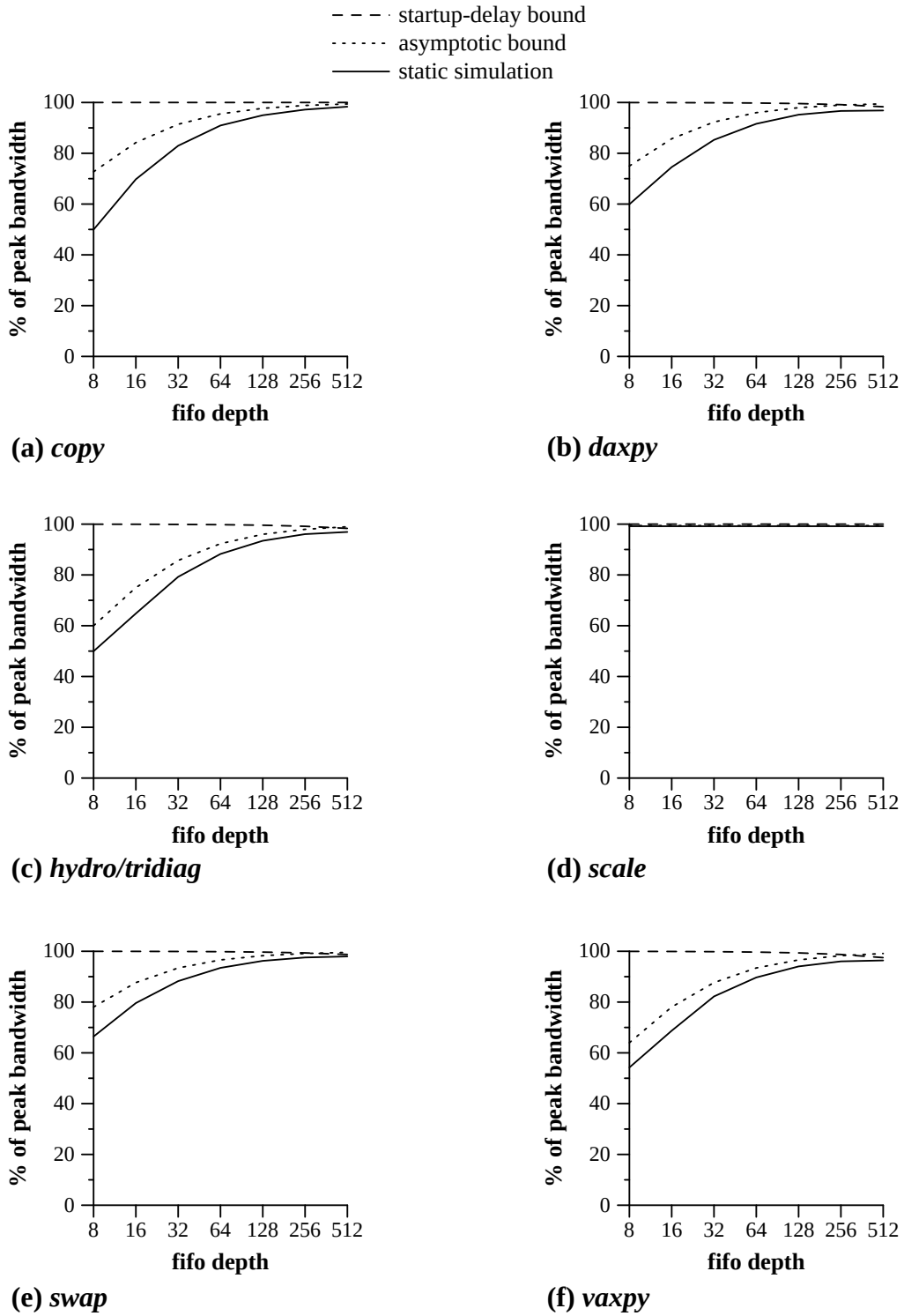
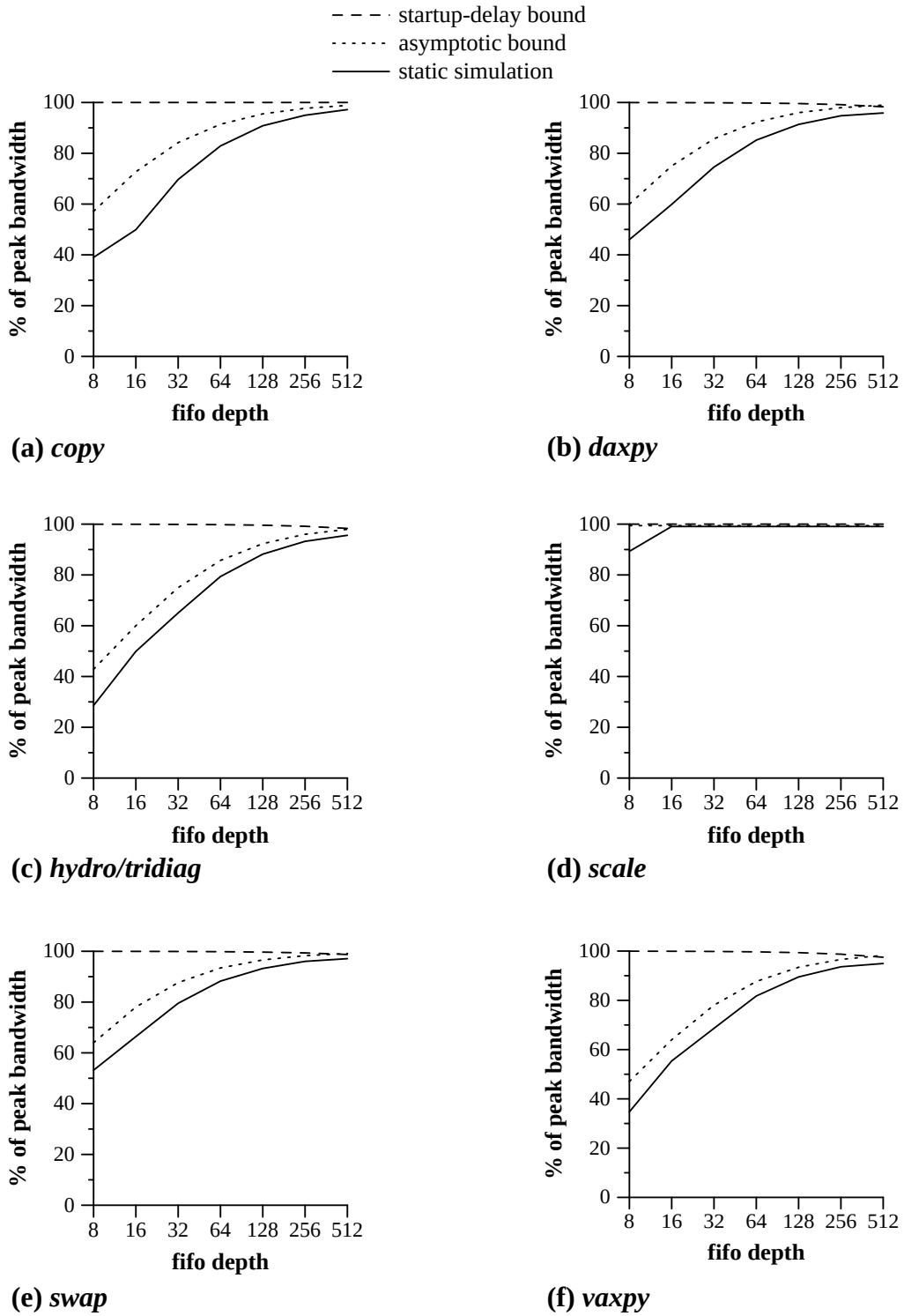


Figure 56 Static Performance for 8 CEs, 32 Banks, 80,000-Element Vectors



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

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

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

Figure 57 Static Performance for 8 CEs, 64 Banks, 80,000-Element Vectors

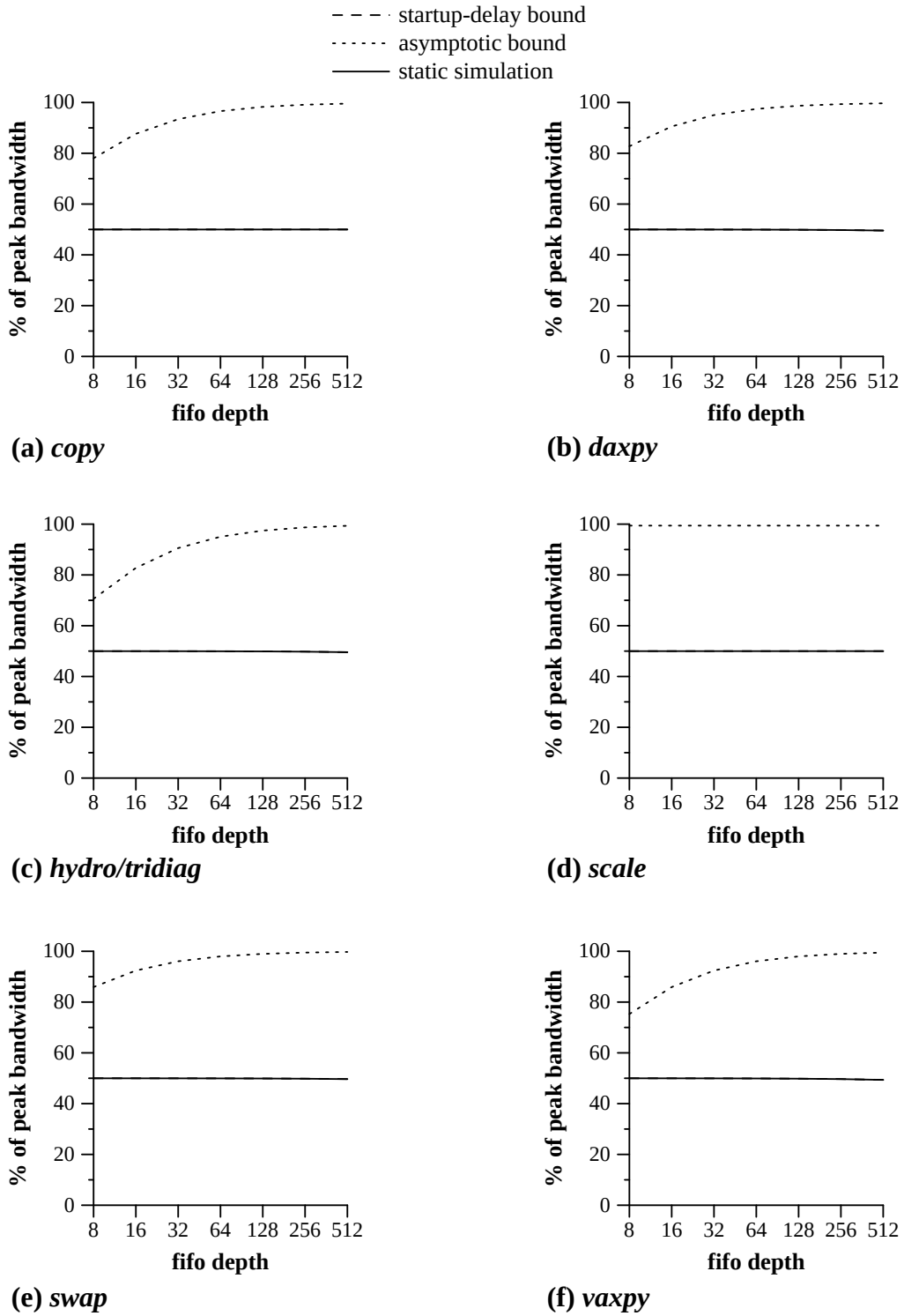


Figure 58 Static Performance for 1 of 2 CEs, 2 Banks, 10,000-Element Vectors

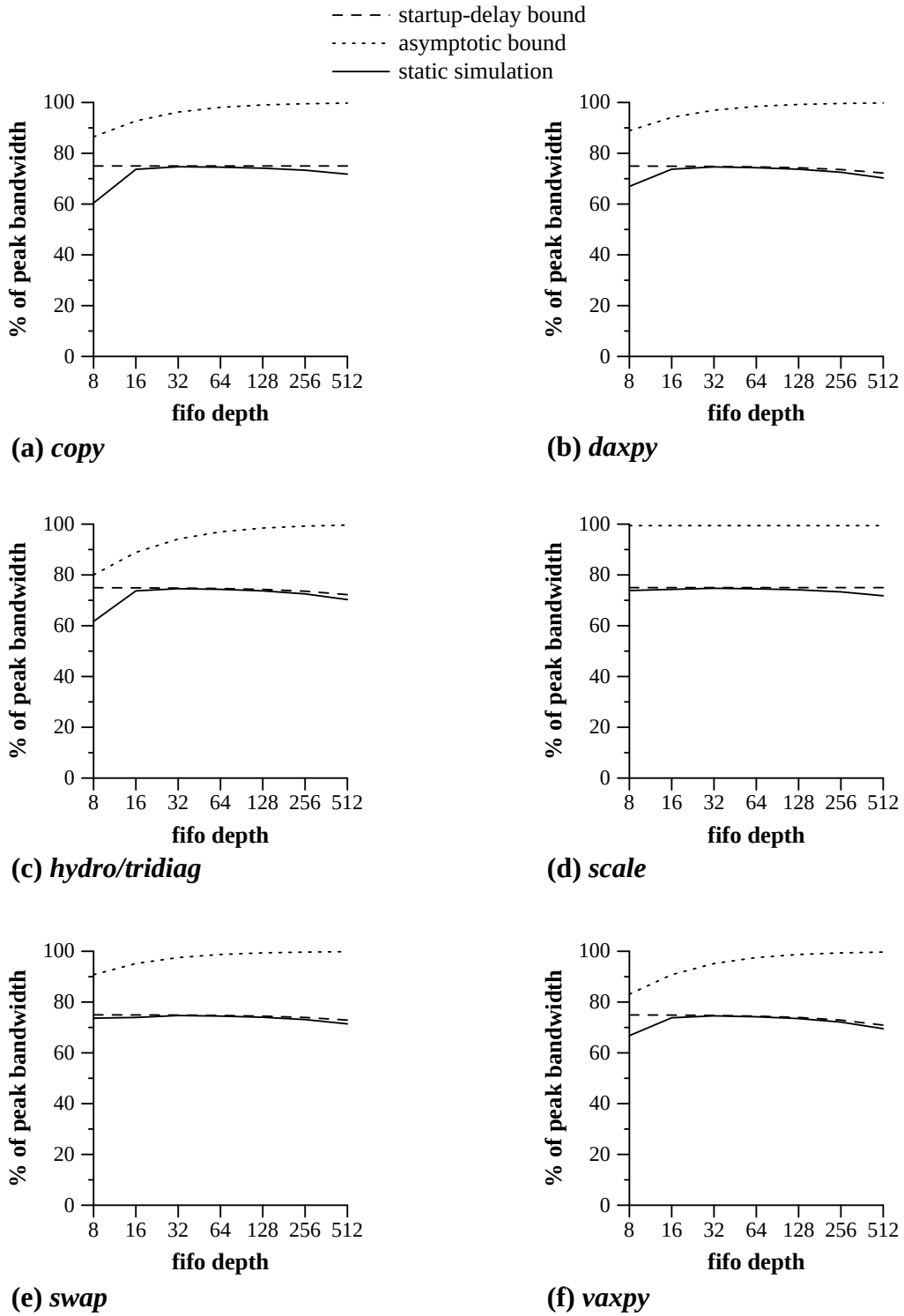


Figure 59 Static Performance for 3 of 4 CEs, 4 Banks, 10,000-Element Vectors

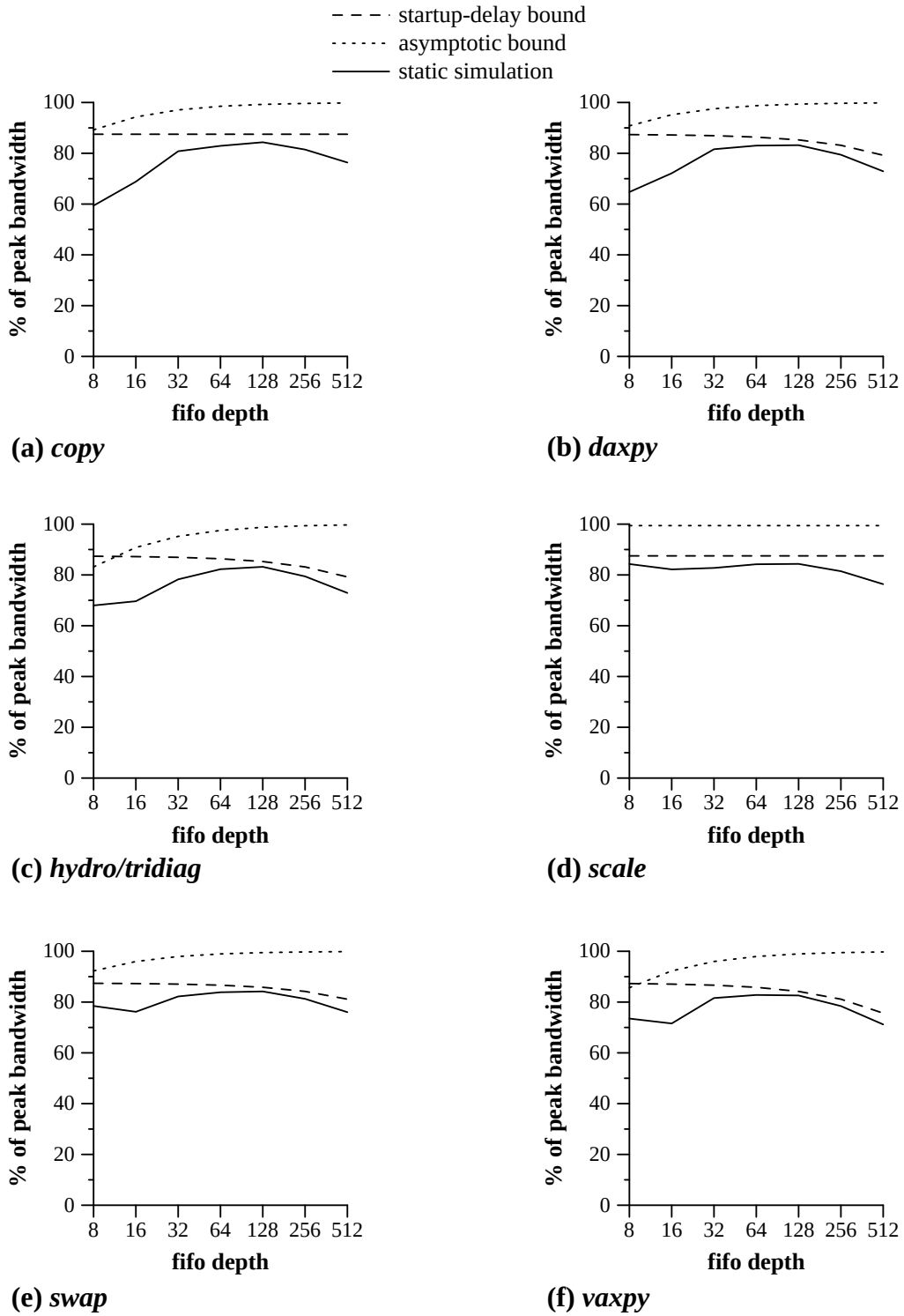


Figure 60 Static Performance for 7 of 8 CEs, 8 Banks, 10,000-Element Vectors

References

[Ben91] Benitez, M.E., and Davidson, J.W., “Code Generation for Streaming: An Access/Execute Mechanism”, Proc. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, April, 1991.

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

[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.

[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.

[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, Technical Report CS-93-54, November, 1993.

[McK93c] McKee, S.A., “Uniprocessor SMC Performance on Vectors with Non-unit Strides”, University of Virginia, Technical Report 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, Technical Report 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, Technical Report CS-93-34.

[McK94c] McKee, S.A., “Dynamic Access Ordering for Symmetric Shared-Memory

Multiprocessors", University of Virginia, Technical Report CS-94-14, April, 1994.

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

[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.

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

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