Hardware Support for Dynamic Access Ordering: Performance of Some Design Options

Sally A. McKee

Computer Science Report No. CS-93-08
August 9, 1993

This work was supported in part by the NSF under contract number MIP-9114110.

Hardware Support for Dynamic Access Ordering:
Performance of Some Design Options

Sally A. McKee
Department of Computer Science
University of Virginia
Charlottesville, VA, 22903
mckee@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* on a uniprocessor system. Our technique combines compile-time detection of memory access patterns with a memory subsystem that decouples the order of requests generated by the processor 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. We present numerous simulation results showing significant speedup on important scientific kernels.

Hardware Support for Dynamic Access Ordering: Performance of Some Design Options

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

1. Increasing Vector Memory Bandwidth

Processor speeds are increasing much faster than memory speeds: microprocessor performance has increased by 50% to 100% *per year* in the last decade, while DRAM performance has risen only 10-15% per year [Kat89]. As a result, memory bandwidth is becoming the limiting performance factor for many applications, particularly scientific computations. Alleviating the growing disparity between processor and memory speeds is the subject of much current research.

Although the addition of cache memory is often a sufficient solution to the memory latency and bandwidth problems in general purpose scalar computing, the vectors used in scientific computations are normally too large to cache, and many are not reused soon enough to benefit much from caching. Furthermore, vectors leave large footprints in the cache. For computations in which vectors are reused, iteration space tiling [Car89, Wol89] can partition the problems into cache-size blocks, but this can create cache conflicts for some block sizes and vector strides [Lam91], and the technique is difficult to automate. Caching nonunit stride vectors leaves even larger footprints, and may actually reduce a computation's effective memory bandwidth by fetching extraneous data.

The traditional scalar processor concern has been to minimize memory latency in order to maximize processor performance. For scientific applications, however, the processor is not the bottleneck, and as processor speeds continue to increase relative to memory speeds,

optimal system performance will leave the processor idle at times. Bridging this performance gap requires changing the way we think about the problem — to maximize bandwidth for scientific applications, we need to minimize *average* latency over a coherent set of accesses.

While many scientific computations are limited by memory bandwidth, they are by no means the only such computations. Any computation involving linear traversals of vector-like data, where each element is typically visited only once during lengthy portions of the computation, can suffer: examples include string processing, image processing and other DSP applications, some database queries, some graphics applications, and DNA sequence matching.

2. Access Ordering

The assumptions made by most memory architectures simply don't match the physical characteristics of the devices used to build them. Memory components are usually assumed to require about the same amount of time to access any random location; indeed, it was this uniform access time that gave rise to the term RAM, or *Random Access Memory.* Many computer architecture textbooks ([Bar92, Hay88, Hwa84, and Man82] among them) specifically cultivate this view. Others skirt the issue entirely [Mac93, Tom90].

Somewhat ironically, this assumption no longer applies to modern memory devices: most components manufactured in the last ten to fifteen years provide special capabilities that make it possible to perform some access sequences faster than others. For instance, nearly all current DRAMs implement a form of page-mode operation [Qui91]. These devices behave as if implemented with a single on-chip cache line, or *page* (this should not be confused with a virtual memory page). A memory access falling outside the address range of the current DRAM page forces a new page to be accessed. The overhead time required

to set up the new page makes servicing such an access significantly slower than one that hits the current page.

Other common devices offer similar features, such as nibble-mode, static column mode, or a small amount of SRAM cache on chip. This sensitivity to the order of requests is exacerbated in several emerging technologies: for instance, Rambus [Ram92], Ramlink, and the new DRAM designs with high-speed sequential interfaces [IEEE92] provide high bandwidth for large transfers, but offer little performance benefit for single-word accesses.

For multiple-module memory systems, the order of requests is important on yet another level: successive accesses to the same memory bank cannot be performed as quickly as accesses to different banks. To get the best performance out of such a system, we must take advantage of the architecture's available concurrency.

A comprehensive, successful solution to the memory bandwidth problem must therefore exploit the richness of the *full* memory hierarchy, both its architecture and its component characteristics. One way to do this is via *access ordering*, which we define as any technique for changing the order of memory requests to increase bandwidth. Here we are especially concerned with ordering a set of vector-like "stream" accesses. For a more thorough discussion of access ordering, see [Moy92, Moy93, McK93a, McK93b].

The performance benefits of doing such static access ordering can be quite dramatic [Moy92, Moy93], but without the kinds of address alignment information that are usually only available at run time, the compiler can't generate the optimal access sequence. The extent to which a compiler can perform this optimization is further constrained by such things as the size of the processor register file. Moreover, the technique is *only* suitable for uniprocessor systems: in a multiprocessor environment, information about other processors' streaming activities is unavailable at compile time. The impact of access ordering on effective memory bandwidth and the limitations inherent in implementing the

technique statically motivate us to consider an implementation that reorders accesses dynamically at run time.

There are a number of other hardware and software techniques that can help manage the imbalance between processor and memory speeds. These include altering the placement of data to exploit concurrency [Gup88], reordering the computation to increase locality, as in "blocking" [Lam91], address transformations for conflict-free access to interleaved memory [Har89, Rau91, Val91], software prefetching data to the cache [Cal91, Kla91, Soh91], and hardware prefetching vector data to cache [Bae91, Fu91, Jou90, Skl92]. For a more detailed discussion of how these schemes relate to dynamic access ordering, see [McK93b]. The main difference between these techniques and the complementary one we propose here is that we *reorder* stream accesses to exploit the architectural and component features that make memory systems sensitive to the sequence of requests.

3. The Stream Memory Controller

The design space of access ordering systems and the rationale for the approach presented here is discussed in [McK93a, McK93b]. The approach we suggest is generally applicable to any uniprocessor computing system, but will be described based on the simplified architecture of Figure 1. Memory is interfaced to the processor 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.

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 registers that the processor uses to specify stream parameters (base address, stride, length, and data size) and high-speed buffers for stream operands. 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.

There are a number of options for the internal architecture of the SBU: here we describe one feasible organization. The set of memory-mapped registers mentioned above provides a processor-independent way of specifying steam parameters. Setting these registers allows the processor to initiate an asynchronous stream of memory access operations for a set of string operands. Data retrieval from the streams (loads) and insertion into streams (stores) may be done in any of several ways; for instance, the SBU could appear to be a traditional cache, or the model could include a set of FIFOs, as illustrated in Figure 1. In this organization, each stream is assigned to one FIFO, which is asynchronously filled from (or drained to) memory by the access/issue logic. 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.

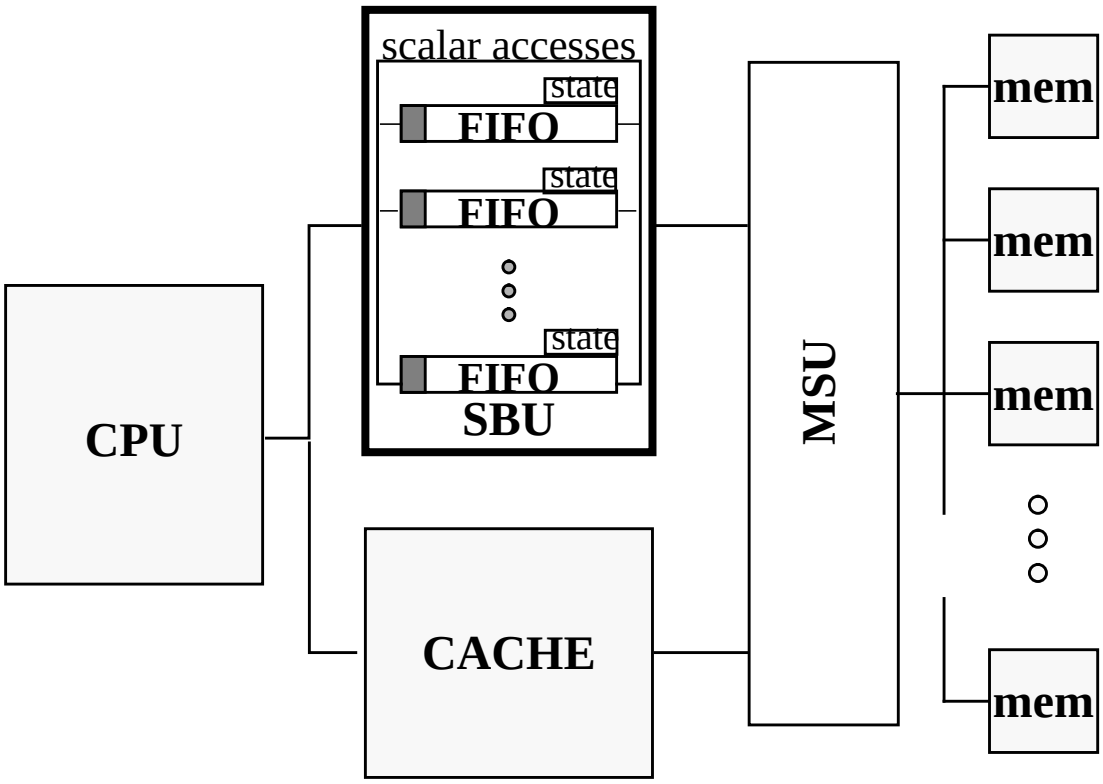


Figure 1 Stream Memory Controller

This organization is both simple and practical from an implementation standpoint: similar designs have been built. In fact, the FIFO organization is almost identical to the “stream units” of the WM architecture [Wul92], or may be thought of as a special case of a decoupled access-execute architecture [Goo85, Smi87]. Another advantage is that this combined hardware/software scheme doesn’t require heroic compiler technology — the compiler need only detect the presence of streams, and Davidson’s streaming algorithm [Dav90] can be used to do this.

Note that we assume the processor can perform non-caching loads and stores so that non-unit stride streams can be accessed without concomitantly accessing extraneous data and wasting bandwidth. While not a common architectural feature, some commercial processors such as the Convex C-1 [Wal85] and Intel i860 [Int91] include such “cache

bypassing". Others, such as the DEC Alpha [DEC92], provide a means of specifying some portions of memory as non-cacheable.

4. Simulation Environment

We have simulated a wide range of SMC configurations, varying

- FIFO depth,
- vector length, stride, and alignment
- dynamic order/issue policy,
- number of memory modules,
- DRAM speed,

The results given here involve the following restrictions:

- All memories modeled here consist of interleaved banks of page-mode DRAMs, where each page is 2K double-words.
- The DRAM page-miss cycle time is four times that of a DRAM page hit, unless otherwise noted.
- NonSMC results are for the "natural" reference sequence for each benchmark, using non-caching loads and stores.
- SMC initialization requires two writes to memory-mapped registers for each stream; this overhead has no significant effect on results, and is not included here.
- We model the processor as a generator of load and store requests only — arithmetic and control are assumed never to be a computational bottleneck. This places the 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.

5. Benchmark Suite

The benchmark kernels used are described in Figure 2. *Daxpy, copy, scale,* and *swap* are from the BLAS (Basic Linear Algebra Subroutines) [Law79, Don79]. These vector and matrix computations occur frequently in scientific computations, thus they have been collected into libraries of highly optimized routines for various host architectures. *Hydro* and *tridiag* are the first and fifth Livermore Loops [McM86], a set of kernels culled from important scientific computations. The former is a fragment of a hydrodynamics computation, and the latter is a tridiagonal elimination computation. *Vaxpy* is a vector *axpy* computation that occurs in matrix-vector multiplication by diagonals; this algorithm is useful for the diagonally sparse matrices that arise frequently when solving parabolic or elliptic partial differential equations by finite element or finite difference methods [Gol93]. *Mul* is a sparse matrix multiply, and *msort* is a merge sort.

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:	$\forall i$	$y_i \leftarrow a_i x_i + y_i$
msort:	*merge sort*	
mul:	*sparse matrix multiply*	

Figure 2 Benchmark Algorithms

Here “axpy” refers to a computation involving some entity *a* times a vector *x*. plus a vector *y.* For *daxpy, a* is a double-precision scalar, so the computation is effectively a scalar times a vector, plus another vector. In the case of *vaxpy, a* is a vector, making the computation a vector times a second vector, plus a third vector.

These benchmarks were selected because they represent access patterns found in real scientific codes, including the inner-loops of blocked algorithms. Note also that these benchmarks constitute a representative subset of all *possible* access patterns for computations involving a small number of vectors (computations requiring more vectors can usually be broken down into several parts, each using only a small number of vectors).

Note that although these computations do not reuse vector elements, they are often found in the inner loops of algorithms that do. Examples include the blocked algorithms of the Level 3 BLAS libraries [Don90], as well as the matrix-multiply by diagonals operation mentioned above (which uses *vaxpy*). Whether or not the vectors are reused has no bearing on SMC performance, although lack of temporal locality greatly diminishes the effectiveness of caching. This is one of the main attractions of dynamic access ordering: we can obtain good memory performance even for computations that do not benefit from caching.

Results for *mul* and *msort* are not addressed here. Our many simulations indicate that the performance curves for the other benchmarks are remarkably similar: this results from the SMC's robust ability to reorder accesses, regardless of the access pattern expected by the processor.

6. Access-Ordering Algorithms

The SMC attempts to exploit the underlying memory architecture to issue accesses in an order that optimizes memory bandwidth. For any memory system composed of interleaved banks of DRAM components, there are at least two facets to this endeavor: taking advantage of the available concurrency among the interleaved banks, and taking advantage of the device characteristics. At each "decision point" (i.e. each available memory bus cycle), the SMC must decide how best to achieve these goals.

The algorithm design space can be divided into two subspaces: algorithms that first choose a bank (*bank-centric* schemes), and algorithms that first choose an access (*access-centric* schemes). In the following discussion, we assume a memory composed of interleaved banks of page-mode DRAMs and a FIFO-based SBU implementation, as depicted in Figure 1.

The Bank-Centric Approach

In these schemes, each bank operates independently, thus each may be on a different DRAM page at any given time. This kind of memory architecture differs slightly from traditional interleaving schemes, where each bank "listens" to the page address for each access, but only one bank responds to the request.

Any bank-centric algorithm for choosing the next access must:

1) select the memory bank(s) to which the next access(es) will be issued, and

2) choose an appropriate access from the pool of ready accesses for each memory bank (this is equivalent to selecting a FIFO to service).

Here 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).

The Access-Centric Approach

These schemes are similar to the bank-centric schemes, except that they perform the two tasks (access selection and bank selection) in the reverse order. Once the FIFO to service has been determined, the selection mechanism chooses an appropriate bank from the set of banks servicing that FIFO.

6.1 Bank Selection

The access selection mechanism must first decide which bank should be accessed next. The possible candidates are those banks that are presently idle. Since there may be fewer banks than potential accesses, it makes sense to determine the set of available banks, and then to consider only accesses to those.

Strategies for selecting banks vary in the number of banks accessed at a time, and in how many banks they consider in their search. At one end of the spectrum lies the exhaustive search strategy — keep looking until the appropriate number of banks is found or no unexamined banks remain — while at the other end of the spectrum only one bank is considered. These schemes must also impose an ordering on the banks to determine which will be considered first.

We have simulated the three *bank-selection* schemes listed in Table 1.

Table 1: Bank Selection Schemes

P	Parallel Access Initiation
R	Round-Robin Selection
T	Token Round-Robin Selection

In the first scheme, *Parallel Access Initiation* (P), we attempt to initiate accesses to all available (non-busy) banks. This greedy algorithm attempts to take full advantage of available concurrency, but is generally impractical to implement, since it requires a separate bus to each bank. Intuitively, it seems that this algorithm should perform at least as well as any other, but this isn't always the case. In general, the interaction between memory bank availability, access initiation, and processor activity is quite complex, and often non-intuitive.

In the *Round-Robin Selection* (R) scheme, we only want to initiate one access, but we consider each bank in turn until we find an available one or have no more banks left. In a

balanced system, where the number of banks is matched to the memory speed, Scheme R essentially staggers the accesses, so that it performs similarly to Scheme P, but with slightly greater latency. The advantage of this algorithm is lower implementation cost, since the bandwidth requirements between the SMC and memory are lower than for Scheme P.

In the last scheme, *Token Round-Robin Selection* (T), we only want to initiate one access, and we only consider the next bank in sequence (if it's busy, we do nothing at the current time). This is the easiest and least expensive to implement of the three algorithms. In spite of T's simplicity, its performance rivals and sometimes exceeds that of Scheme P and Scheme R.

For the R and T approaches, the most reasonable strategy is to start with the next bank in sequence after the bank to which the last access was initiated. Starting with a fixed bank each time would cause some banks to be under-used: accesses to those banks would effectively have lower priority.

6.2 Fifo Selection

The FIFO-selection algorithms vary in sophistication, ranging from those that use all available information to decide what to do next, to those that do the easiest (and quickest) thing they can.

Look for the Page Hit?

Some algorithms first look for an access that hits the bank's current DRAM page. Others simply choose the next FIFO in round-robin order, regardless of whether the next access from that FIFO hits the current page.

Look for Best Candidate?

If an algorithm that looks for a page hit can't find one, how does it choose the next access? Some look for a "best" candidate based on how full (empty) the read (write) FIFOs are:

since we know we're going to incur the page-miss overhead, we'd like to amortize that cost over as many page-hits as possible, hence we want to choose a FIFO for which there will be many accesses to the new DRAM page. Other algorithms simply choose the next FIFO in sequence when they can't find a page-hit.

Global versus Local Status Information

When trying to decide which FIFO is "best" to service next, the algorithm may consider the total contents of the FIFO (this is the global view), or it may restrict itself to just the portion of the FIFO for which the current bank is responsible, referred to as a *subFIFO* (this is the local view). Some algorithms require that a FIFO (subFIFO) meet a certain "threshold" in order to be considered for service: for instance, an algorithm might require that a read FIFO (subFIFO) be at least half empty before it can be considered among the best candidates for the next access. The rationale for this sort of restriction springs from the overhead involved in accessing a new DRAM page: any time we must switch DRAM pages, we would like to amortize the cost of that miss over as many accesses as possible. If there are sufficiently few ready accesses to a given page, it may be worthwhile to wait until the processor has generated more accesses to that page (by removing elements from the read FIFO or writing elements to the write FIFO) than to pay the page overhead now. In the event that these algorithms find no valid candidates, they either choose the next FIFO in sequence, or do nothing until the next decision-making time.

Where to Start the Search?

There are several possibilities for prioritizing the FIFOs. We could consider them in random order; we could impose a fixed order, always considering a given FIFO first; we could give priority to reads (or writes); we could start with the last FIFO the selected bank serviced; or we could start with the last FIFO *any* bank serviced. The latter two options seem most fair and reasonable from an implementation standpoint. The first of these encourages different banks to be working on different FIFOs, while the second encourages

several banks to be working on the same FIFO. It is not intuitively obvious which of these will yield better performance.

We chose a set of ten FIFO-selection algorithms spanning the design space and conducted numerous simulations for each combination of bank- and FIFO-selection schemes. Table 2 lists these schemes along with an explanation of how they work.

Table 2: FIFO-Selection Algorithms

1	look for page hit; if none, choose fullest write/emptiest read subFIFO search round-robin, starting with last FIFO accessed by current bank
2	look for page hit; if none, choose fullest write/emptiest read subFIFO that's at least 1/2 full/empty; if none, choose next access found search round-robin, starting with last FIFO accessed by current bank
3	look for page hit; if none, choose fullest write/emptiest read subFIFO that's at least 1/2 full/empty; if none, do nothing search round-robin, starting with last FIFO accessed by current bank
4	look for page hit; if none, choose next access found search round-robin, starting with last FIFO accessed by current bank
5	choose next access search round-robin, starting with last FIFO accessed by current bank
6	look for page hit; if none, choose fullest write/emptiest read subFIFO search round-robin, starting with last FIFO accessed by any bank
7	look for page hit; if none, choose fullest write/emptiest read FIFO search round-robin, starting with last FIFO accessed by current bank
8	look for page hit; if none, choose fullest write/emptiest read FIFO search round-robin, starting with last FIFO accessed by any bank
9	look for page hit; if none, choose next access found search round-robin, starting with last FIFO accessed by any bank
10	choose next access search round-robin, starting with last FIFO accessed by any bank

6.3 The Algorithms Simulated

Each pairwise combination of bank-selection and FIFO-selection algorithms (P1 through T10) describes a particular bank-centric ordering scheme. In addition to these, we simulated two naive access-centric ordering schemes. Scheme A1 is exceedingly simple:

the SMC looks at each FIFO in round-robin order, issuing accesses for the same FIFO stream while

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

Scheme A2 is similar, except it incorporates the notion of a threshold into the decision whether to continue servicing the same FIFO: accesses that incur page-misses will only be issued to the current FIFO if it is empty enough (for a read FIFO) or full enough (for a write FIFO), otherwise each FIFO in sequence is evaluated according to the same criteria. If none is found to meet the threshold, no access is initiated at that time.

7. Results

Section 7.1 presents detailed results for each combination of bank-selection schemes and the first five FIFO-selection algorithms. Each of these fifteen access ordering schemes was run on a variety of memory systems (a single-bank system, as well as interleaved systems of two, four, and eight banks) in which DRAM page-misses cost four times as much as DRAM page hits. Simulation results for the remaining five FIFO-selection algorithms are extremely similar, thus we omit discussing them at length, but instead provide a brief summary of their comparative performance in Section 7.2. We then demonstrate in Section 7.3 how the addition of an SMC affects bandwidth as page-miss costs change in relation to page-hit costs. Section 7.4 summarizes our findings.

7.1 Simulation Results

These results are for the seven benchmark algorithms in Figure 2, run on long (10,000-element), medium (100-element), and short (10-element) vectors. The *hydro* and *tridiag* benchmarks share the same access pattern, thus their results for these simulations are identical, and are presented together in each figure.

We have chosen 10,000 elements as our “long” vectors, although much longer vectors (on the order of millions of elements) certainly exist in practice. These vectors are long enough that SMC startup transients become insignificant, thus we do not expect performance for million-element vectors to be materially different. There’s another argument in favor of choosing a length of 10,000 as opposed to one million: consider the effects of context switches when using an SMC in a multiprogrammed environment. Let us consider a hypothetical RISC system running at 50MHz, and assume that on average it executes one instruction per 20ns clock cycle. If such a system incurred a context switch about one hundred times a second, it could execute roughly 500,000 instructions between context switches. In other words, we could reasonably expect to perform on the order of 10,000 iterations of an inner loop (up to 50-instructions) between context switches. Thus our choice of “long” vector length is appropriate: long enough that startup transients have essentially no effect on performance, and short enough that the vectors represent an amount of work that might reasonably be accomplished between context switches.

Figure 3 and Figure 4, Figure 7, Figure 10, …, Figure 55 show SMC performance for long vectors as a function of FIFO depth and number of memory banks (available concurrency) compared to the analogous nonSMC systems. For these simulations, all vectors are aligned to begin in the same bank. Figure 5, Figure 8, …, Figure 56 depict SMC performance for medium vectors compared to the analogous nonSMC memory systems, but here the vectors used for the nonSMC results have a better alignment: the i^{th} vector begins in bank (i mod n), where n is the total number of banks. Figure 6, Figure 9, …, Figure 57 illustrate SMC performance on very short (10-element) vectors. NonSMC performance is as depicted in the long or medium vector graphs, depending on vector alignment; those lines are omitted from these graphs for clarity.

7.1.1 Group 1 — Algorithms P1, R1, and T1

Algorithm P1

Recall that this ordering algorithm attempts to initiate an access to each idle bank at every available bus cycle. For each memory bank *b*, it examines the FIFOs in round-robin order, beginning with last FIFO for which an access to *b* was initiated. If it finds an access that hits the current DRAM page, it issues that access. If no accesses for the bank hit the current DRAM page, then an access is issued for the FIFO requiring the most service from *b*. Figure 3 through Figure 6 depict P1's performance.

Figure 3 and Figure 4 show SMC performance for vectors of 10,000 elements as a function of FIFO depth and number of memory banks. Most of the results presented here will be as in Figure 3, where performance is given as a percentage of normalized peak bandwidth. Results for memory systems with a greater number of modules represent a percentage of a larger bandwidth. The bottom curves in Figure 3 depict the bandwidth attained by the analogous nonSMC systems. On the *daxpy* benchmark, for example, an SMC system with two memory banks achieves 97.8% of peak bandwidth, compared to 18.7% for a nonSMC system. In general, SMC systems with deep FIFOs achieve in excess of 94% of peak bandwidth for all benchmarks and memory configurations. The only exception is *tridiag*, which attains 91% of peak on the four-bank system, and 85% of peak with eight banks. Even with FIFOs that are only sixteen double-words deep, the SMC systems consistently deliver over 80% of the attainable bandwidth. Again, the *tridiag* benchmark is the lone exception; here SMC systems with sixteen-deep FIFOS achieve over 73% of peak.

The performance differences between *tridiag* and the other kernels stem from its access pattern: it uses three vectors, but accesses each only once per iteration. *Vaxpy* also involves three vectors, but it splits the *y* vector into two streams: read and write. This reuse gives it a lower percentage of page misses for the SMC to amortize. Similarly, *copy* and *scale* are distinguished by the presence in the latter of a vector that is both read and written.

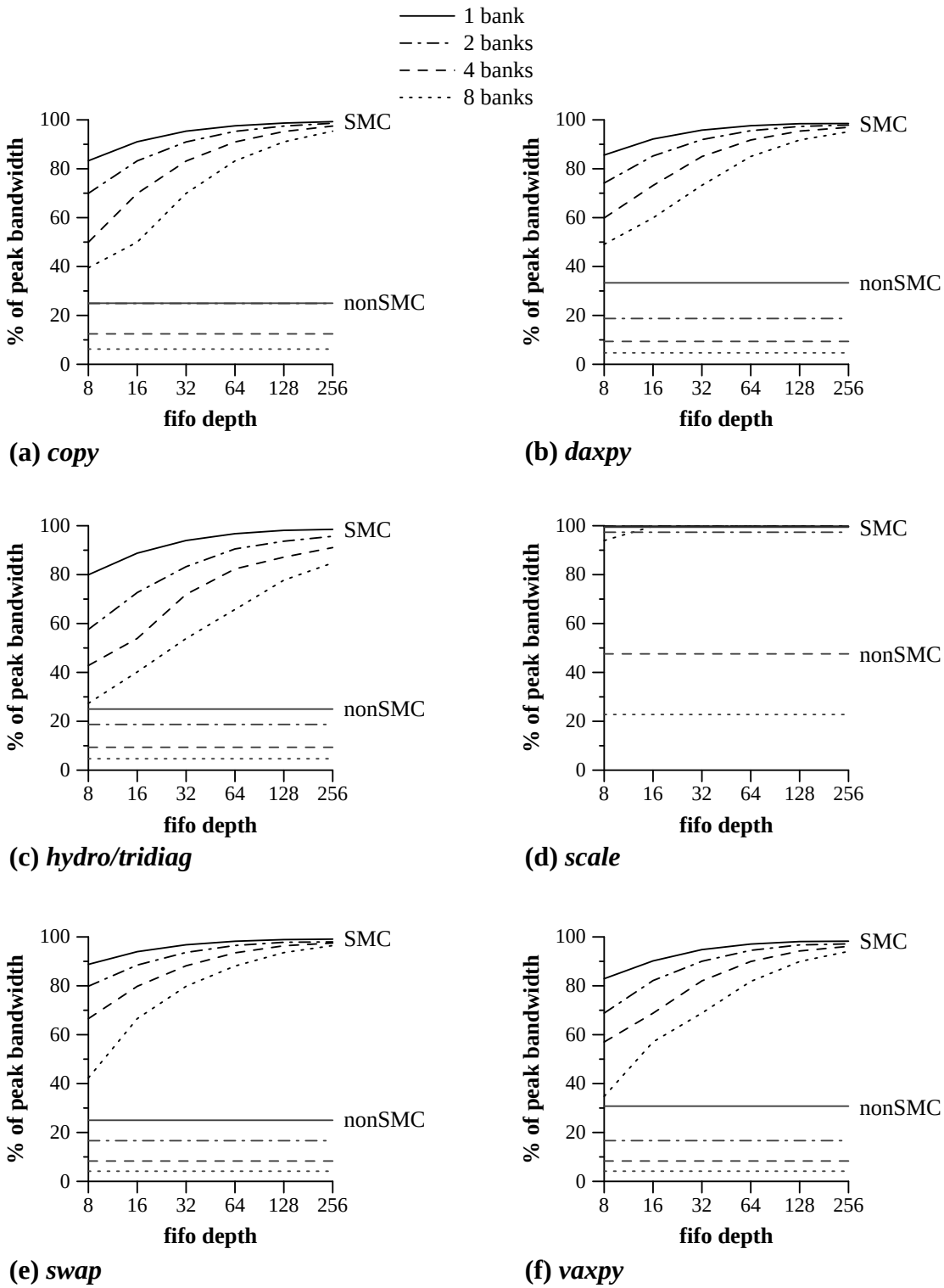


Figure 3 P1 — Long Vector Performance

Note that increasing the number of banks reduces *relative* performance, a somewhat counter-intuitive and deceptive effect. This is due in part to our keeping both the peak memory system bandwidth and the DRAM page-miss/hit delay ratio constant. Thus, the eight-bank system has four times the DRAM page-miss latency of the two-bank system. Although the percentage of peak bandwidth delivered for the architectures with greater interleaving is smaller, the total bandwidth is much larger. If, alternatively, we hold the page-miss cycle time of the memory components constant, decrease the page-hit cycle time, and assume a faster bus, the peak bandwidth of the total system increases proportionally to the number of banks. Figure 4 illustrates SMC performance on the *hydro* benchmark under these circumstances. Performance is given as a percentage of the peak bandwidth of a single-bank memory system; gray horizontal lines indicate peak bandwidth for each architecture. Our benchmark achieves a noticeably lower percentage of total bandwidth for the four- and eight-bank architectures. Increasing the number of banks decreases the total number of accesses to each bank, thus page-miss costs are amortized over fewer accesses.

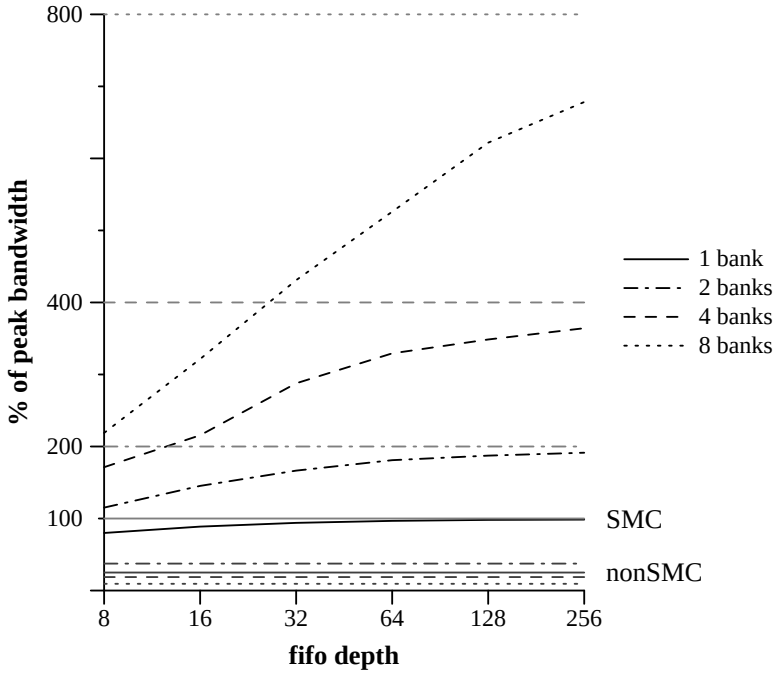


Figure 4 *hydro* Long Vector Performance When Bandwidth Scales With Interleaving

Performance of nonSMC systems is independent of vector length. Since these systems employ no dynamic access ordering, the number of requests issued and the resulting percentage of total bandwidth obtained are constant for each loop iteration. This is true of any system in which access issue is determined at compile time, including those that use prefetching.

Figure 5 depicts the results of simulating selection algorithm P1 on our benchmarks using vectors of 100 elements. These SMC results depict the net effect of two competing performance factors. With deeper FIFOs, DRAM page misses are amortized over a larger number of total accesses, which can increase performance. At the same time, the processor has to wait longer to complete its first loop iteration while the SMC prefetches numerous operands to be used in the following loop iterations. This can decrease performance, as evidenced by the tail-off beyond depth-32 FIFOs. Optimum FIFO depth could, and probably should, be run-time selectable in the SMC, since it is so closely related to stream length.

Lack of dynamic ordering renders the performance of nonSMC systems particularly sensitive to vector placement. In the graphs depicting long-vector SMC performance, the vectors are aligned so that they all compete for the same bank on each iteration; this has little effect on SMC performance (because it reorders requests), but it prevents the nonSMC systems from taking advantage of the potential concurrency. In order to illustrate the effects of alignment on bandwidth, the nonSMC results presented for medium-length vectors represent starting addresses with staggered alignment: the i^{th} vector in the pattern begins in bank (i mod n), where n is the number of banks. In spite of the more favorable alignment, nonSMC *daxpy* performance is limited to 30.0% of total bandwidth for a two-bank memory; *hydro*, *swap*, and *vaxpy* are limited to 18.8%, 40.0%, and 25.0%, respectively. Since *scale* uses only one vector, its performance is unchanged.

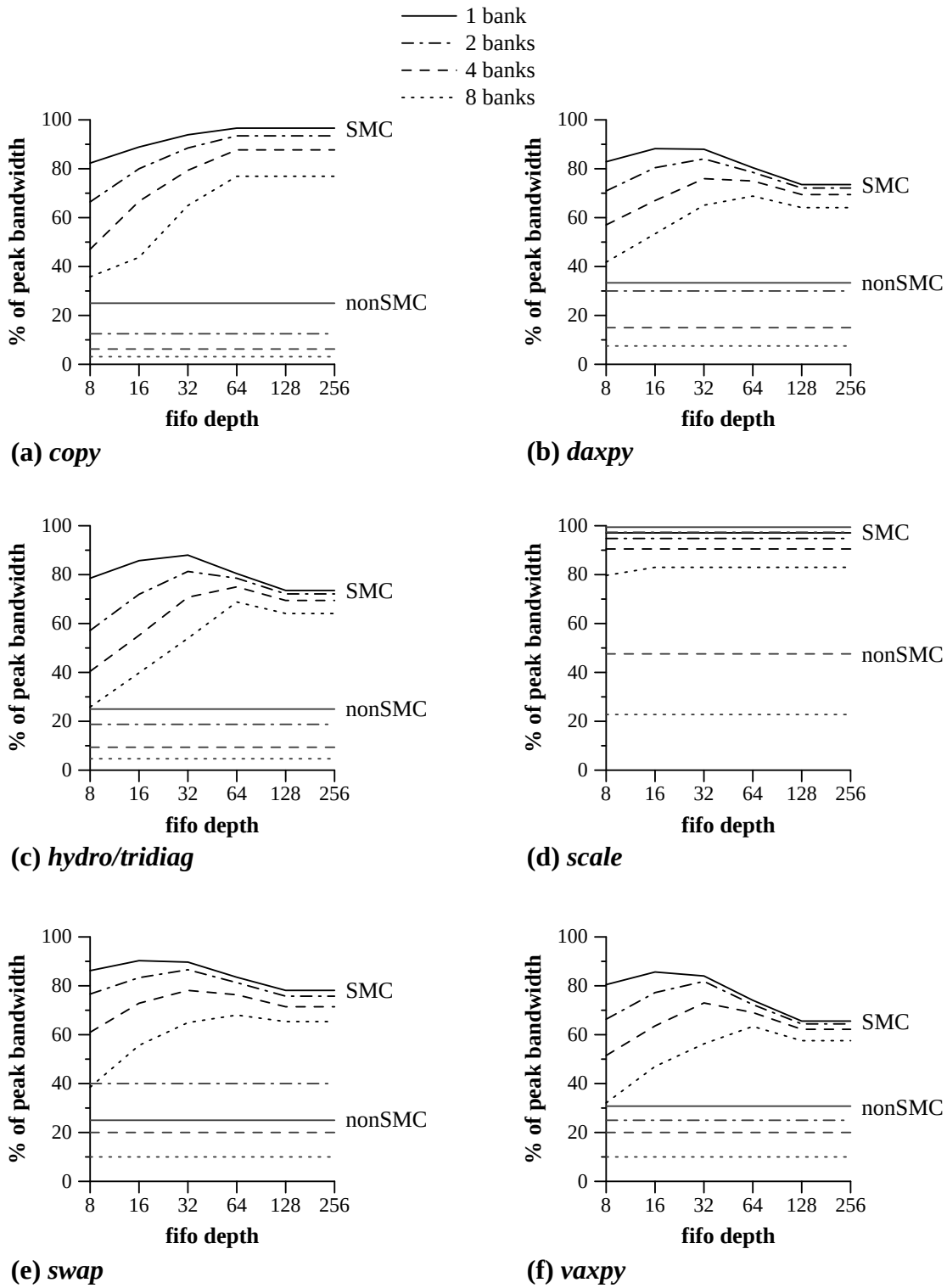


Figure 5 P1 — Medium Vector Performance (for better nonSMC alignment)

Note that for a memory system with eight banks, eight-deep FIFOs are inadequate. For a stride-one vector, each bank will be responsible for servicing only one FIFO position, which severely limits the SMC's ability to amortize DRAM page-miss costs. The SMC's memory access pattern for each bank in this case is almost the same as that generated by the processor, hence performance tends to sink towards that of a nonSMC system. Note that even when the SMC can't take advantage of page-mode accesses, it nonetheless prefetches reads and buffers writes, thus it still offers some performance advantages. In general, the greater the concurrency inherent in the memory system, the deeper the SMC's FIFOs need to be in order to amortize each bank's page-miss overhead.

Figure 6 illustrates SMC performance on very short (10-element) vectors. Performance improvements are not as dramatic as for longer vectors, for there are very few accesses over which to amortize page-miss costs. Nonetheless, short vector computations benefit significantly from an SMC. As noted above, nonSMC performance is as depicted in Figure 3 or Figure 5, depending on vector alignment.

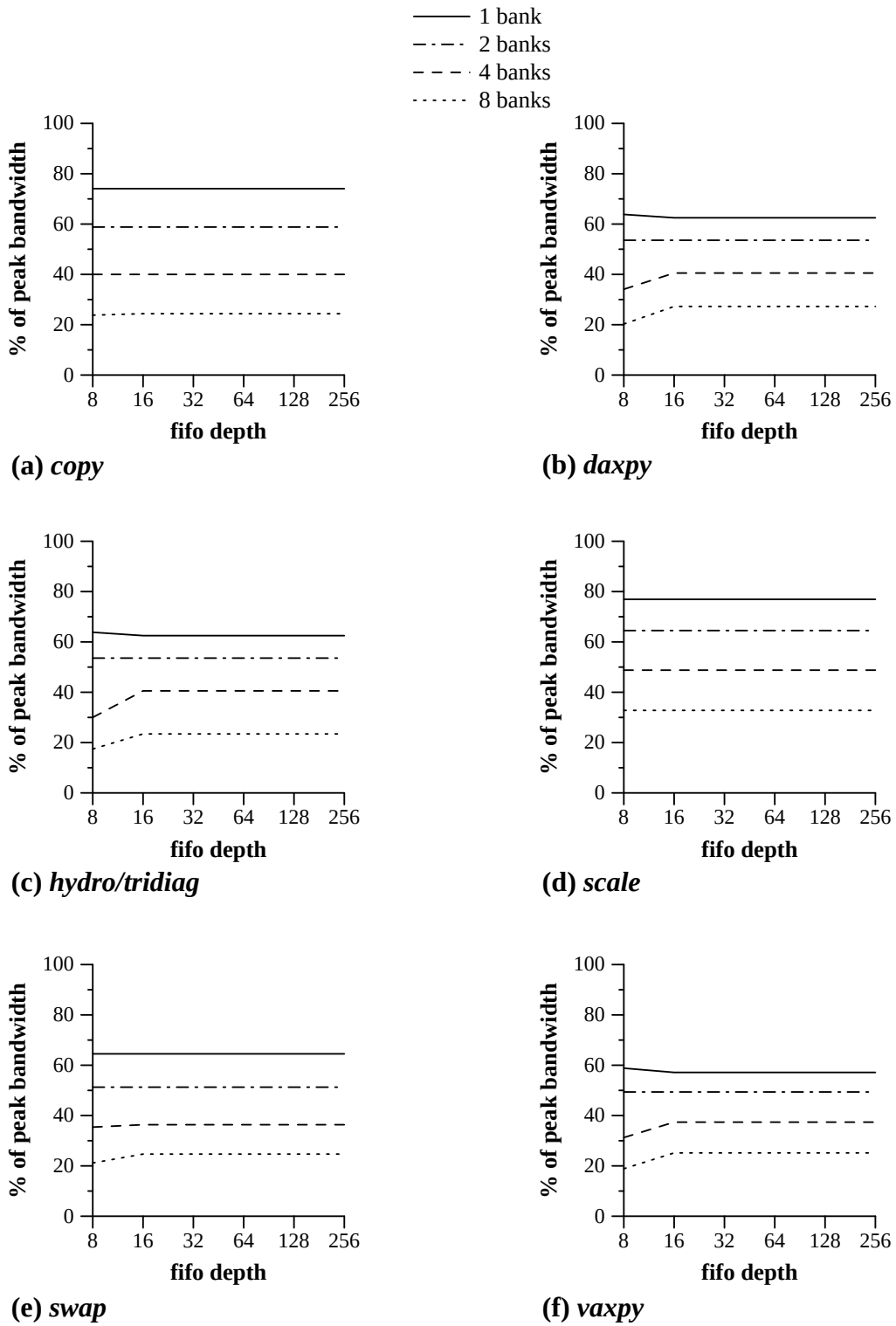


Figure 6 P1 — Short Vector Performance

Algorithm R1

This greedy algorithm is identical to P1, except that only one access may be issued during any one bus cycle. The algorithm examines the banks in round-robin order, beginning with the bank following the one to which the most recent access was made. It attempts to initiate an access (according to the scheme described for P1, above) for the first idle bank it finds. Figure 7 through Figure 9 depict R1's performance.

All three bank-selection schemes perform identically for all benchmarks on a single-bank memory system. Note that for this FIFO-selection scheme, R1's performance is extremely similar to that of algorithm P1 — for systems with two and four banks, performance is identical. For SMC systems with eight banks, performance of the two schemes differs only for very shallow FIFOs, where the SMC is unable to take advantage of page hits.

In fact, performance of *all* the R algorithms is remarkably similar to that of the P algorithms. This stems from the design of the SBU. In the parallel scheme, there is a separate bus to each memory bank, allowing the SMC to initiate several accesses at a time. The SBU in the SMC simulated here can only process one data value at a time, however. This is due to the fact that the FIFOs must be dual-ported in order to allow simultaneous access by both the processor and the MSU; implementing an efficient FIFO to allow more than two simultaneous accesses would be much more difficult, and would consumer substantially more chip real estate. Thus read accesses completing simultaneously are effectively serialized, since all but one of them is delayed until the next cycle. Likewise, the SMC can only write one value each bus cycle. This has the effect of staggering the initiation of accesses to the different banks, so that the parallel algorithms end up behaving much like the greedy round-robin approaches. In light of these limitations, a parallel access-initiation scheme would afford substantial performance benefit only if the SBU were able to process several data values at once (or if it processed them serially, but with a cycle time much faster than that of the memory buses).

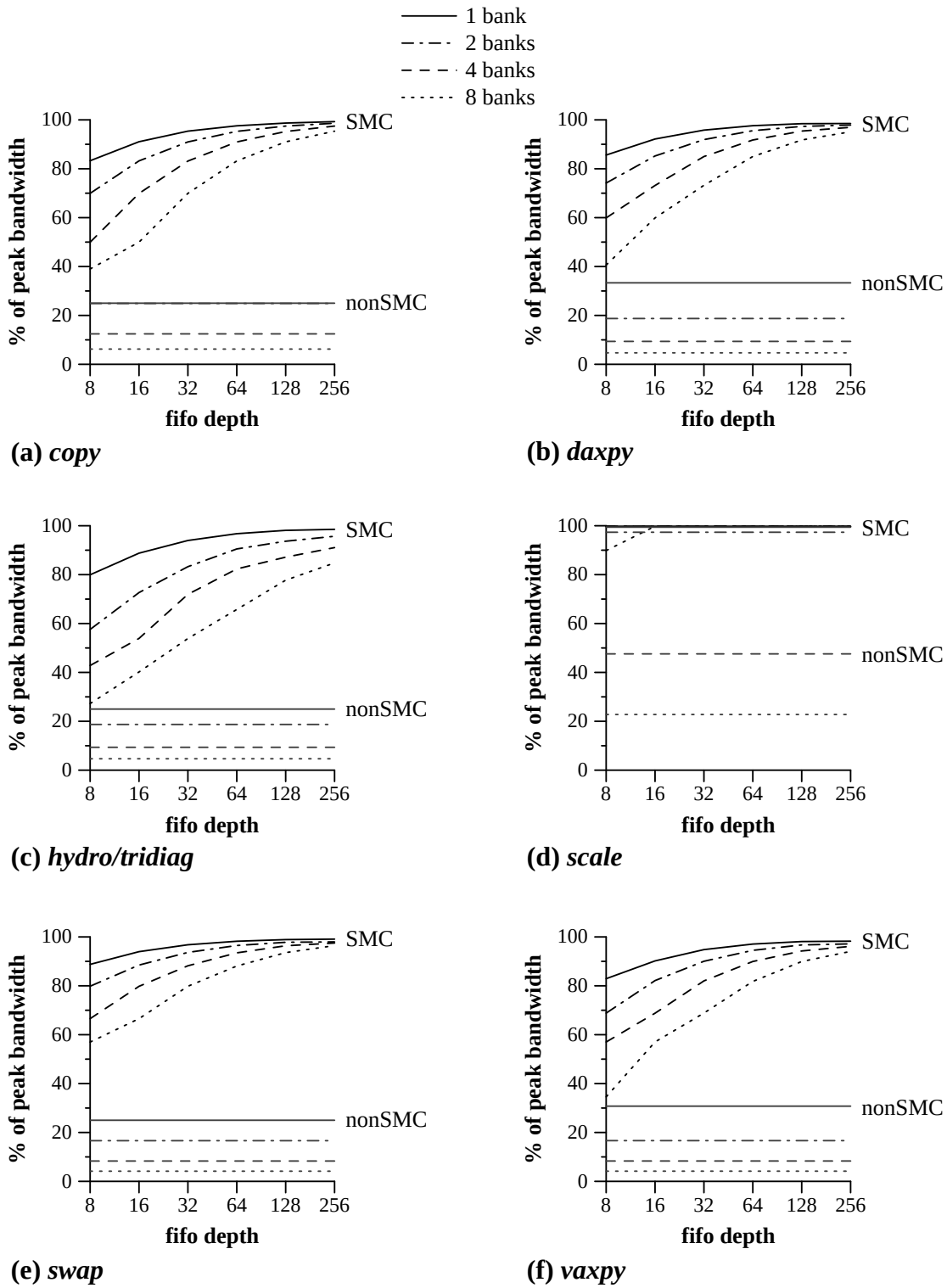


Figure 7 R1 — Long Vector Performance

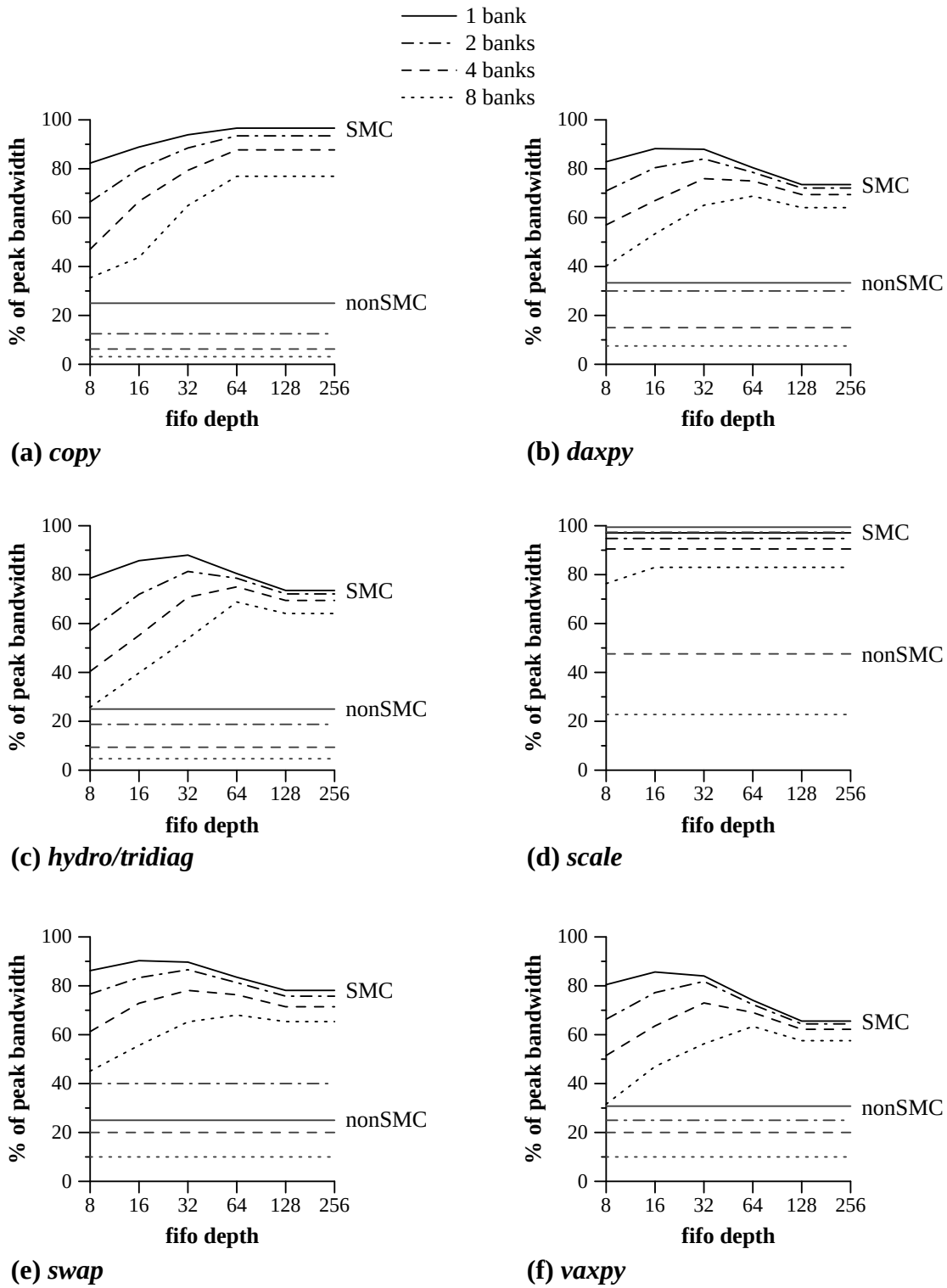


Figure 8 R1 — Medium Vector Performance (for better nonSMC alignment)

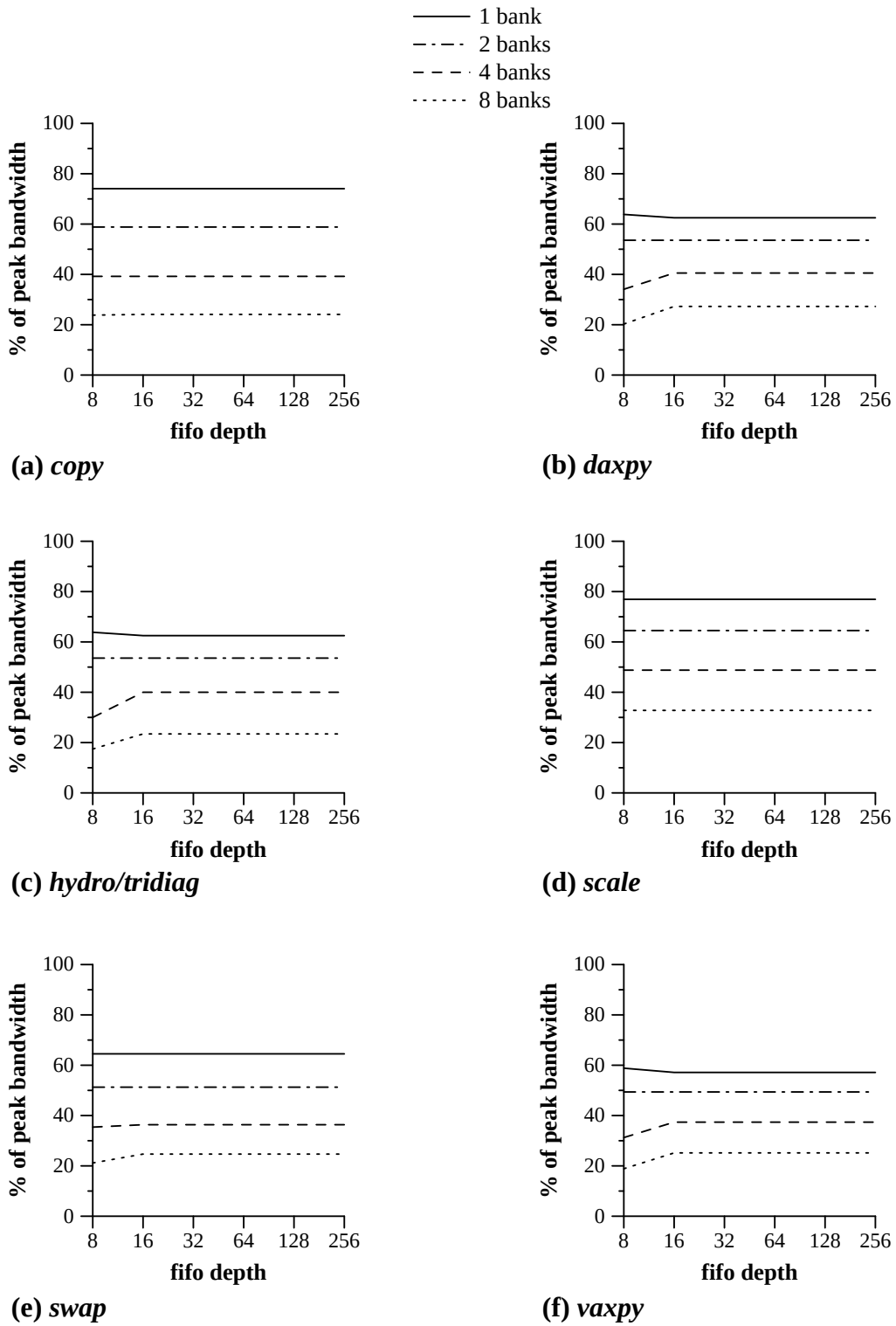


Figure 9 R1 — Short Vector Performance

Algorithm T1

Like P1, Algorithm T1 issues at most one access each bus cycle. Instead of considering each idle bank in turn when attempting to initiate an access, T1 only considers the next bank (in round-robin order from the last bank considered). If that bank is busy, or if no ready access to it exists, then no access is initiated at the current time. Figure 10 through Figure 12 depict T1's performance.

Again, the performance curves are very similar to those for P1 and R1, with results for all but the shallowest FIFOs differing by less than 1% of attainable bandwidth. Results for FIFOs that are only eight double-words deep vary by more than 15% of attainable bandwidth, but only for the eight-bank memory system, where the SMC cannot take advantage of page hits. Note that Algorithm T1 slightly outperforms the other two for some benchmarks.

For short vectors (performance for which is depicted in Figure 12), Algorithm T1 delivers a higher percentage of peak bandwidth for the *scale* and *vaxpy* benchmarks run on a memory system with two banks, although the margin is only a few percent. For instance, on the *scale* computation, Algorithm T1 achieves 36.4% of the peak bandwidth on an eight-bank system, whereas Algorithm R1 reaches only 32.8%. The same benchmark on a two-bank architecture yields 69.0% of peak for Algorithm T1, as opposed to 64.5% for Algorithm R1.

The trends that we have seen among the performances of the P, R, and T bank-selection schemes are present for all groups of algorithms simulated, but there is simply too much data to make meaningful comparisons between all ordering algorithms. Since Scheme T is the most reasonable from an implementation standpoint, we focus most of our discussion on ordering algorithms employing this strategy, and we use Algorithm T1 as a basis of comparison for performance of the other algorithms.

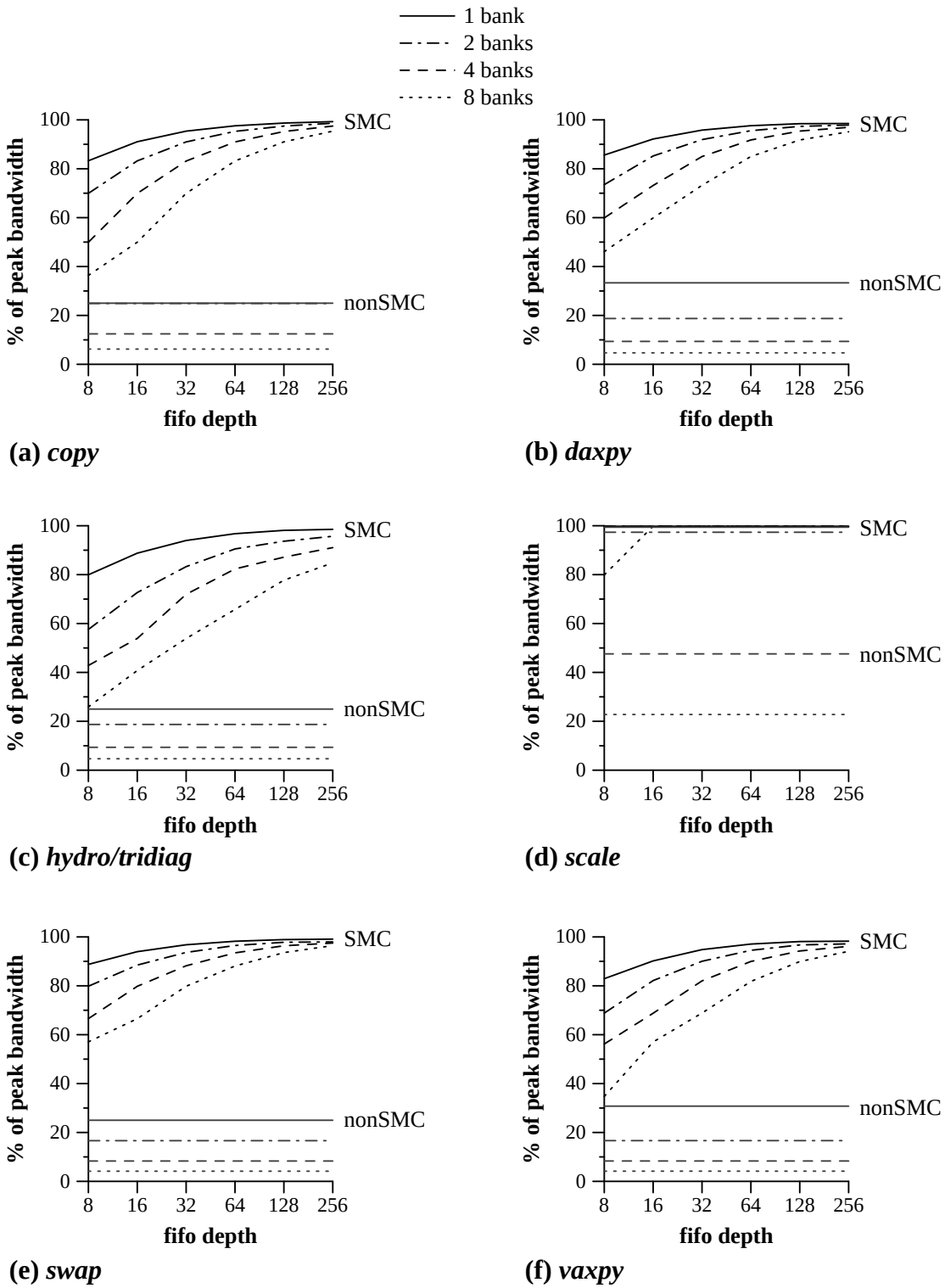


Figure 10 T1 — Long Vector Performance

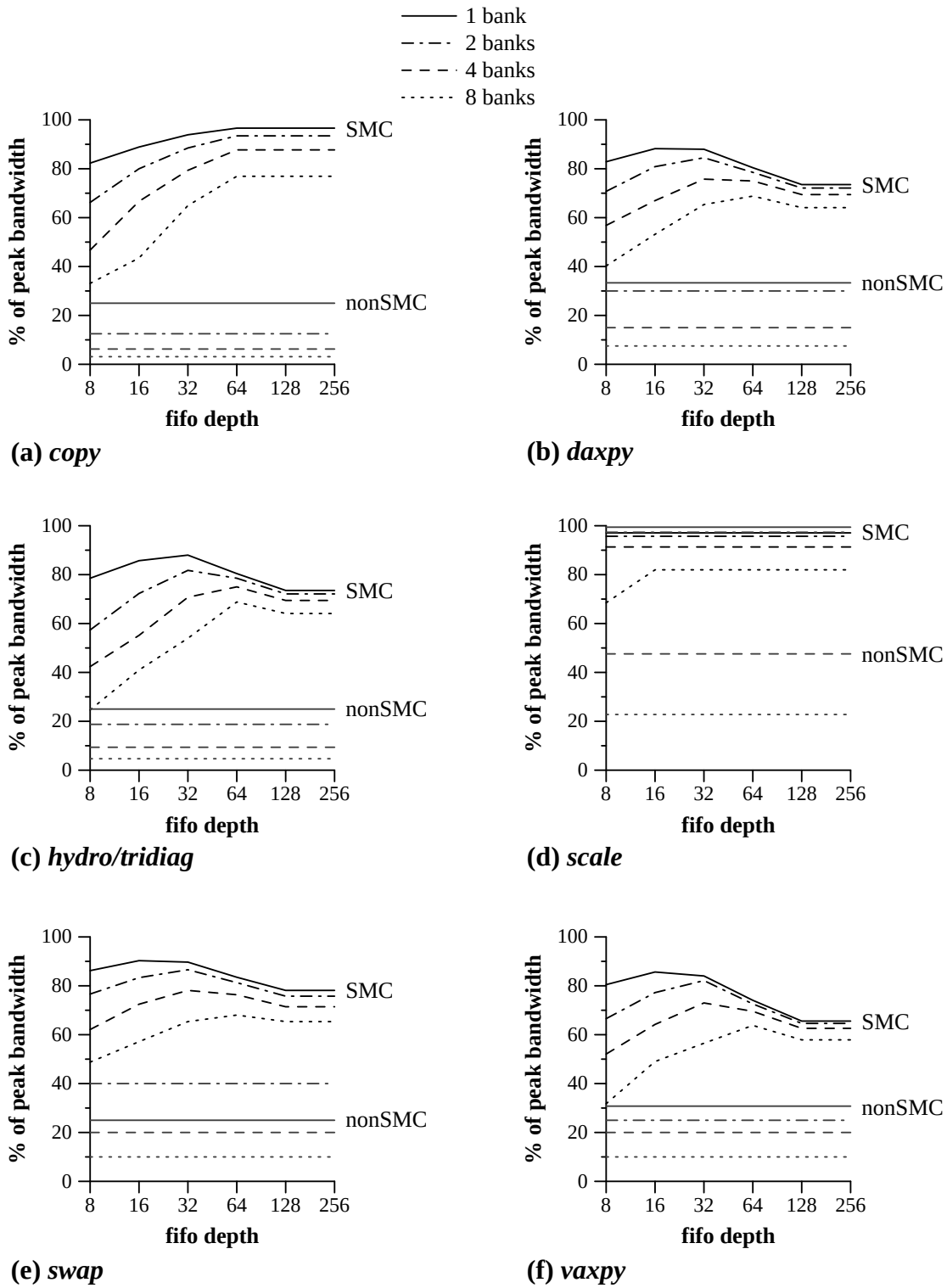


Figure 11 T1 — Medium Vector Performance (for better nonSMC alignment)

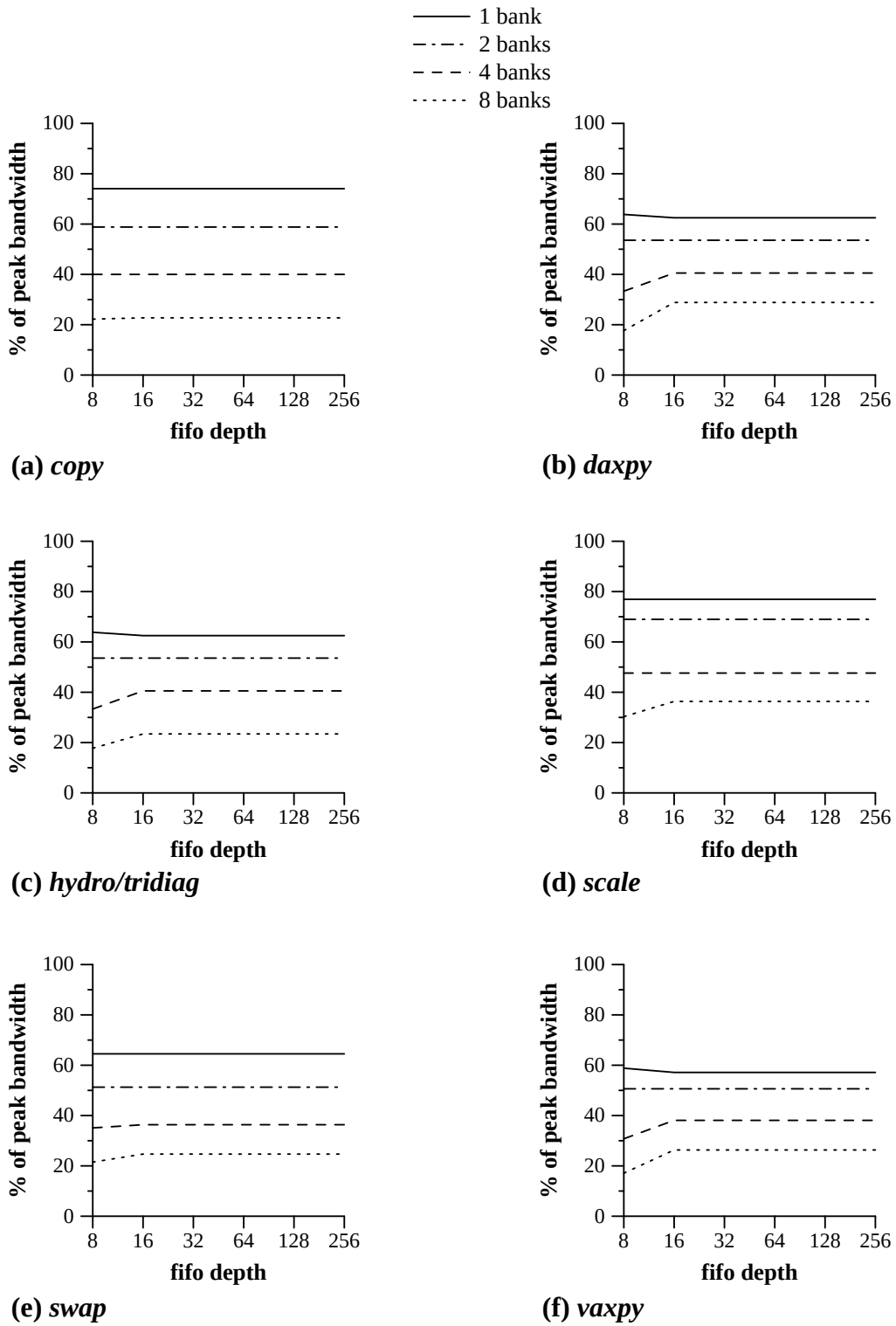


Figure 12 T1 — Short Vector Performance

All the SMC results presented thus far have been for vectors aligned such that corresponding elements of the vectors reside in the same memory bank. This placement degrades the memory performance of nonSMC systems, for it generates bank conflicts and can cause thrashing behavior with respect to DRAM pages. Since the SMC reorders accesses to take advantage of the memory system's available bandwidth, it is relatively insensitive to operand placement and alignment. To illustrate this, Figure 13 through Figure 15 depict SMC performance for Algorithm T1 using the same vector alignment as for the nonSMC results in Figure 5, Figure 8, and Figure 11. Recall that in this alignment, the i^{th} vector in the pattern begins in bank (i mod n), where n is the number of banks. NonSMC results in Figure 13 are as in Figure 10, where vectors are aligned to begin in the same bank; nonSMC results in Figure 14 use the alignment just describe for this set of SMC experiments, are thus are the same as in Figure 11. Since *swap* is unaffected by alignment, results for that benchmark are identical to the corresponding T1 results in Figure 10 through Figure 12.

The differences in performance are summarized in Table 3. Table entries are obtained by subtracting the performance numbers from Figure 10, Figure 11, and Figure 12 from the corresponding results in Figure 13, Figure 14, and Figure 15. A negative entry indicates that the first (single bank) alignment yielded better performance. Values of magnitude greater than 1% are rounded to the nearest tenth. For entries of lesser magnitude, the table contains only the sign of the difference. Blank entries indicate that differences, if any, are less than one hundredth of one percent; note that such entries comprise most of the table. The largest differences occur for memory systems with many banks, especially with shallow FIFOs, where the lack of buffer space prevents the SMC from effectively amortizing page-miss costs. Differences for shorter vectors, although not included here, are smaller still.

Table 3: T1 Long Vector Performance Differences for Two Vector Alignments

Differences in Attained Percentage of Peak Bandwidth (staggered minus single-bank)							
benchmark	memory banks	FIFO depth					
		8	16	32	64	128	256
copy	1						
	2						
	4						
	8		+				
daxpy	1			-	+	-	+
	2	+	-	+	-	-	-
	4	-4.0	+	-	-	+	-
	8	-12.8	-4.0	+	-	-	-
hydro	1	+					
	2	-1.9	-1.5	-			
	4	+	+4.4	-1.8	+	-2.4	-
	8	-	+3.0	-3.7	-1.8	+5.2	-
swap	1		+	-			+
	2	-	+	+	+	+	+
	4	-	-	-		-	+
	8		-	-1.2			
vaxpy	1	+	-	+	+	+	+
	2	+	+	-	+	+	+
	4	-	+	+	-	+	+
	8	-4.0	+	+	-	+	+

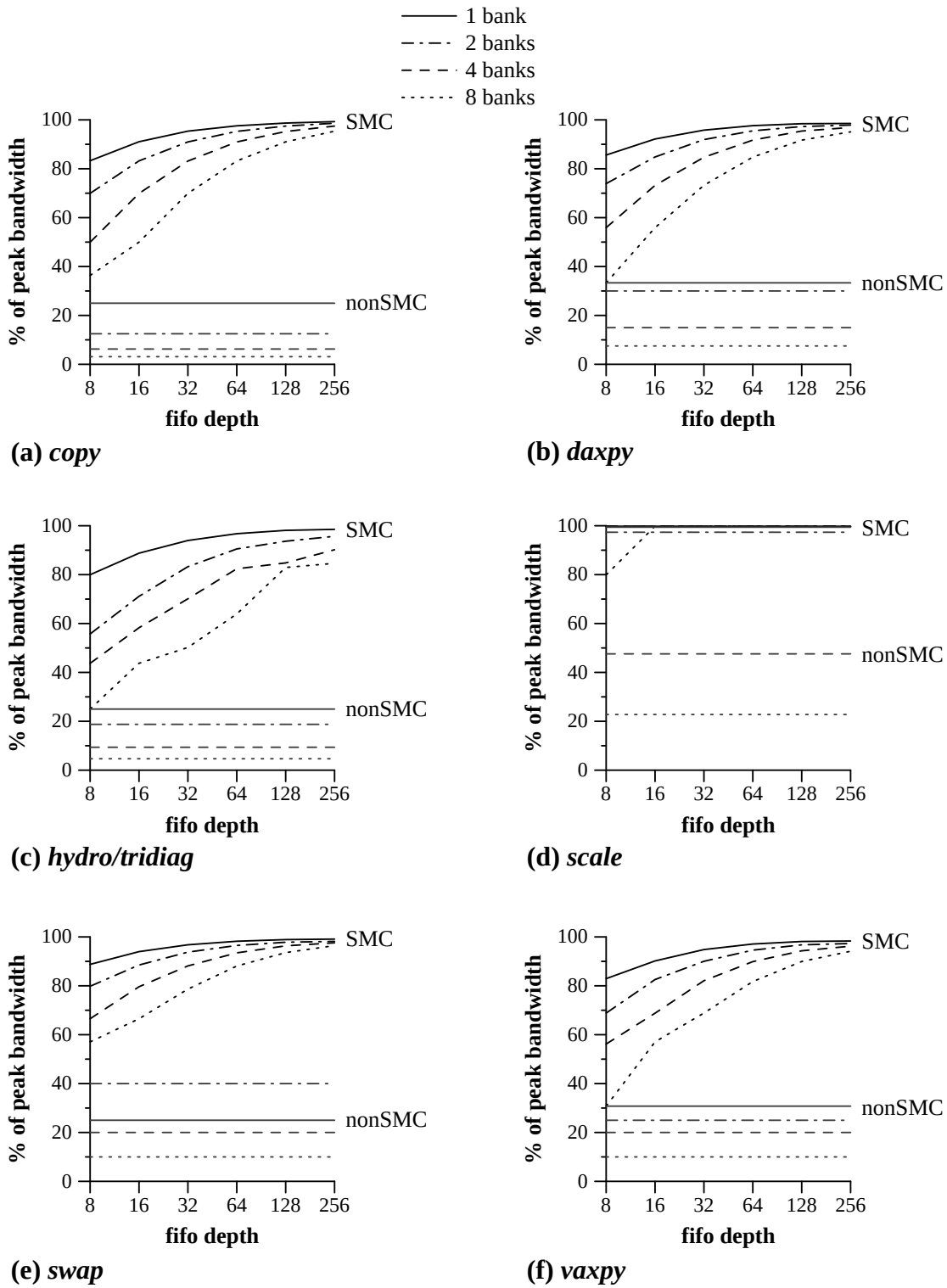


Figure 13 T1 — Long Vector Performance for a Different SMC Vector Alignment

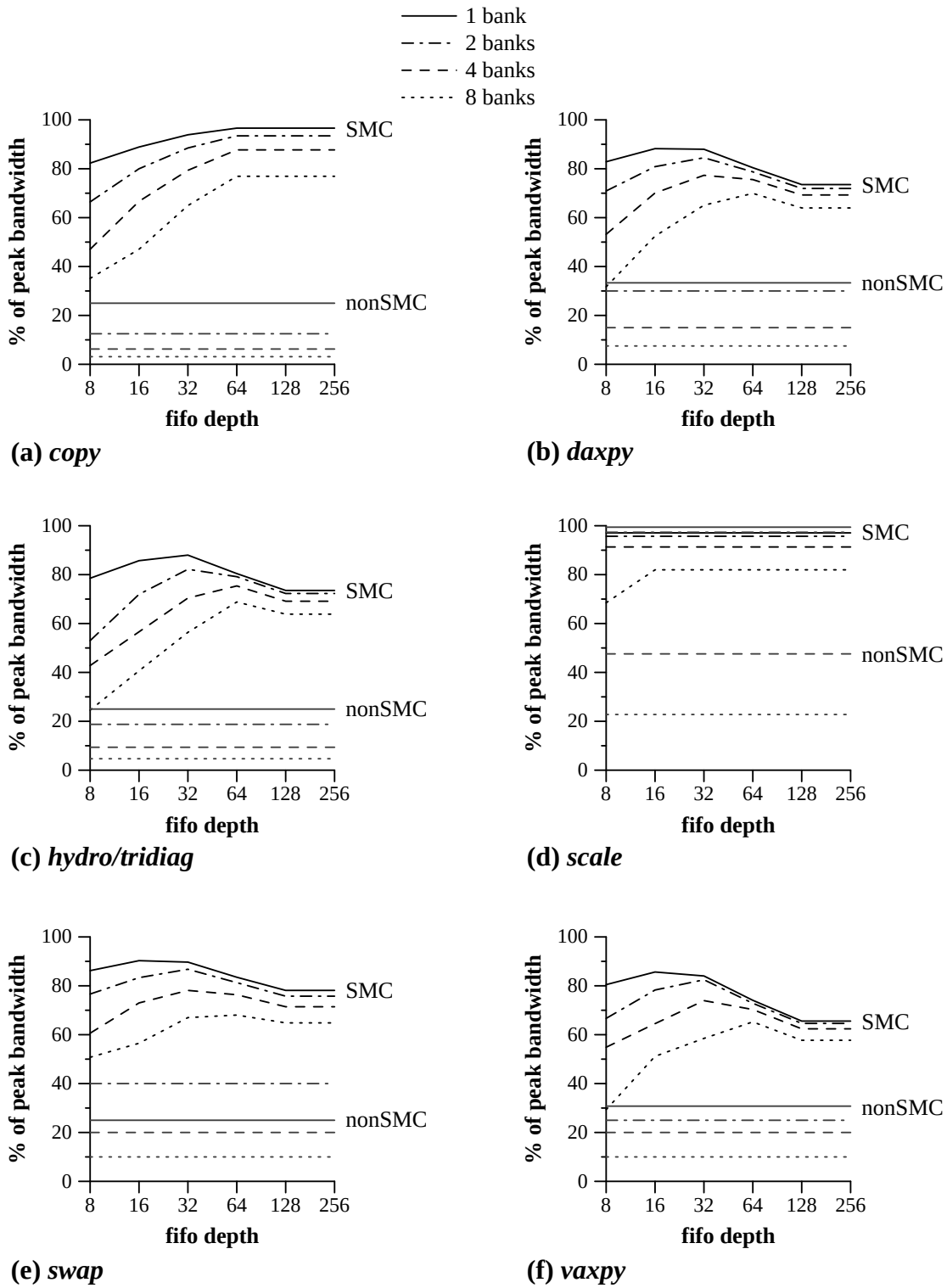


Figure 14 T1 — Medium Vector Performance for a Different SMC Vector Alignment

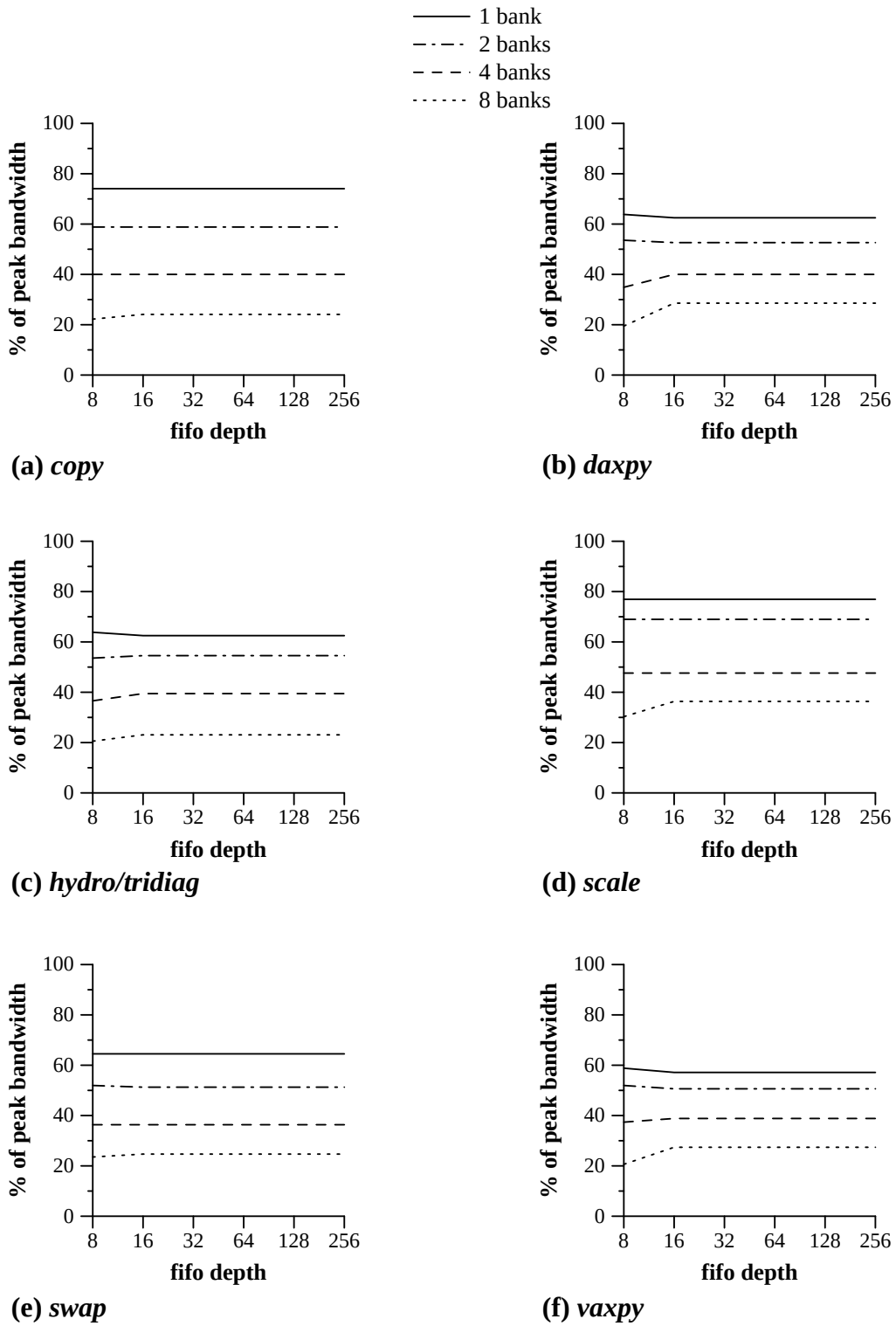


Figure 15 T1 — Short Vector Performance for a Different Vector Alignment

7.1.2 Group 2 — Algorithms P2, R2, and T2

These algorithms are similar to those described in Section 7.1.1, except that they incorporate the notion of a threshold of required service. For each memory bank *b* selected by the access-initiation scheme (P, R, or T), the FIFO-selection algorithm examines the FIFOs in round-robin order, beginning with last FIFO for which an access to *b* was initiated. If it finds an access that hits the current DRAM page, it issues that access. If no accesses for the bank hit the current DRAM page, then it looks for an access from a FIFO containing at least *n* / 2 ready accesses, where *n* is the number of FIFO positions that map to bank *b*. If a FIFO requiring the appropriate amount of service is found, an access is initiated. If no such FIFO exists, the algorithm defaults to using the next FIFO (following the one for which the most recent access to bank *b* was initiated), attempting to initiate an access for it.

The performance of the Group 2 algorithms is depicted in Figure 16 through Figure 24. Performance is extremely similar to that of the corresponding algorithm from Group 1, generally differing by less than 1% of peak bandwidth. The only exception is the *hydro* benchmark. For medium-length vectors, FIFOs of depth sixty-four, and an eight-bank memory, Group 1 beats Group 2 by almost 4% of peak, yet for a two-bank system with FIFOs half that depth, the Group 2 algorithms represent a performance gain of over 2% of peak. For longer vectors, the differences are magnified, and the effect of the threshold is erratic. For the four- and eight-bank memories, Group 2 performance varies from 0.3% of peak worse to 6.7% better (most FIFO depths gain at least 4% of peak), and there is no clear trend in the variations in performance. For *hydro* on very short vectors, Group 1 beats Group 2 by 5.2% of peak for very shallow FIFOs on a two-bank memory system.

The fact that the threshold has relatively little effect on the performance for most benchmarks suggests that when a DRAM page change is necessary, the FIFO requiring the most service either meets the threshold or happens to be the default selection.

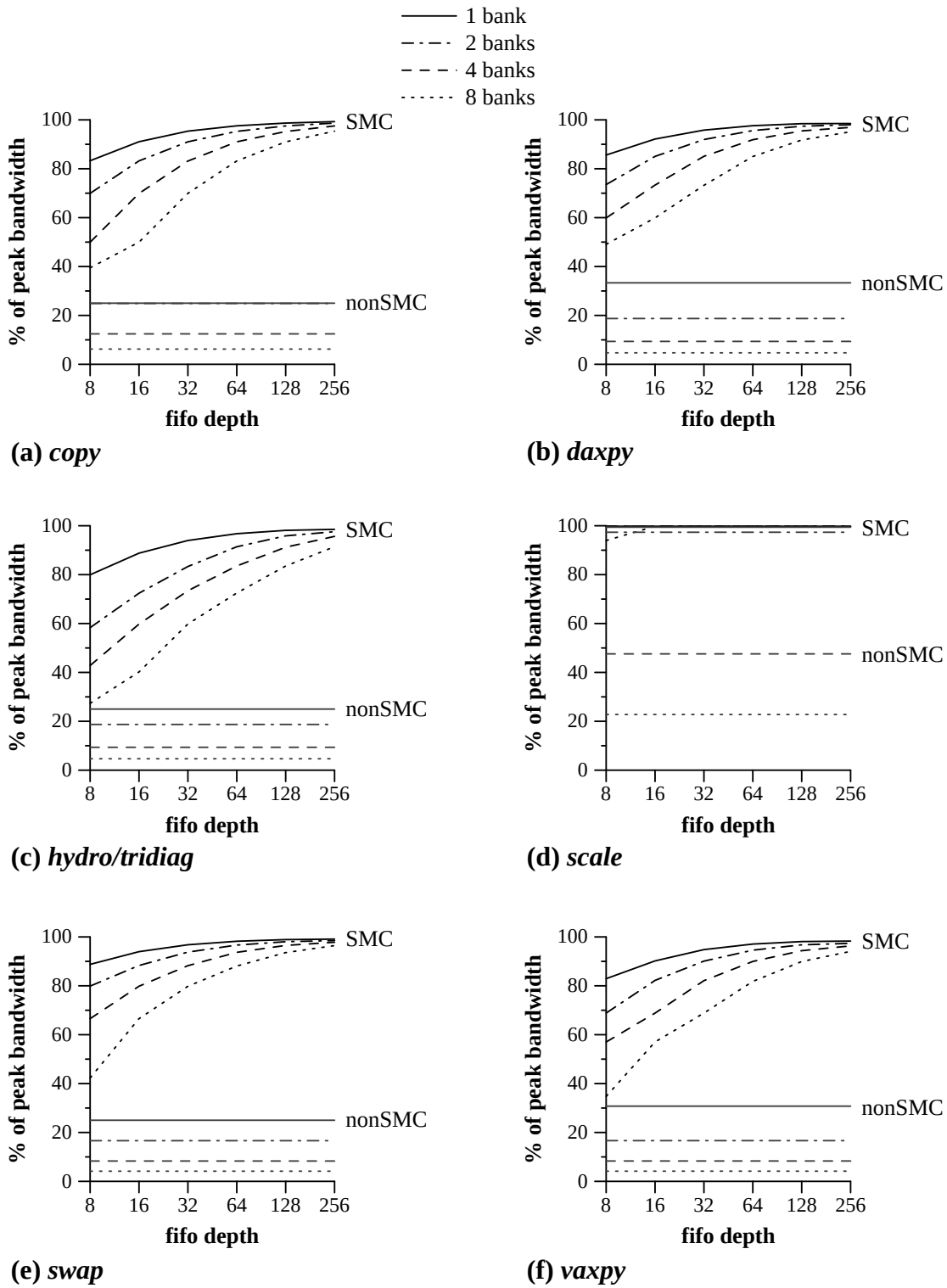


Figure 16 P2 — Long Vector Performance

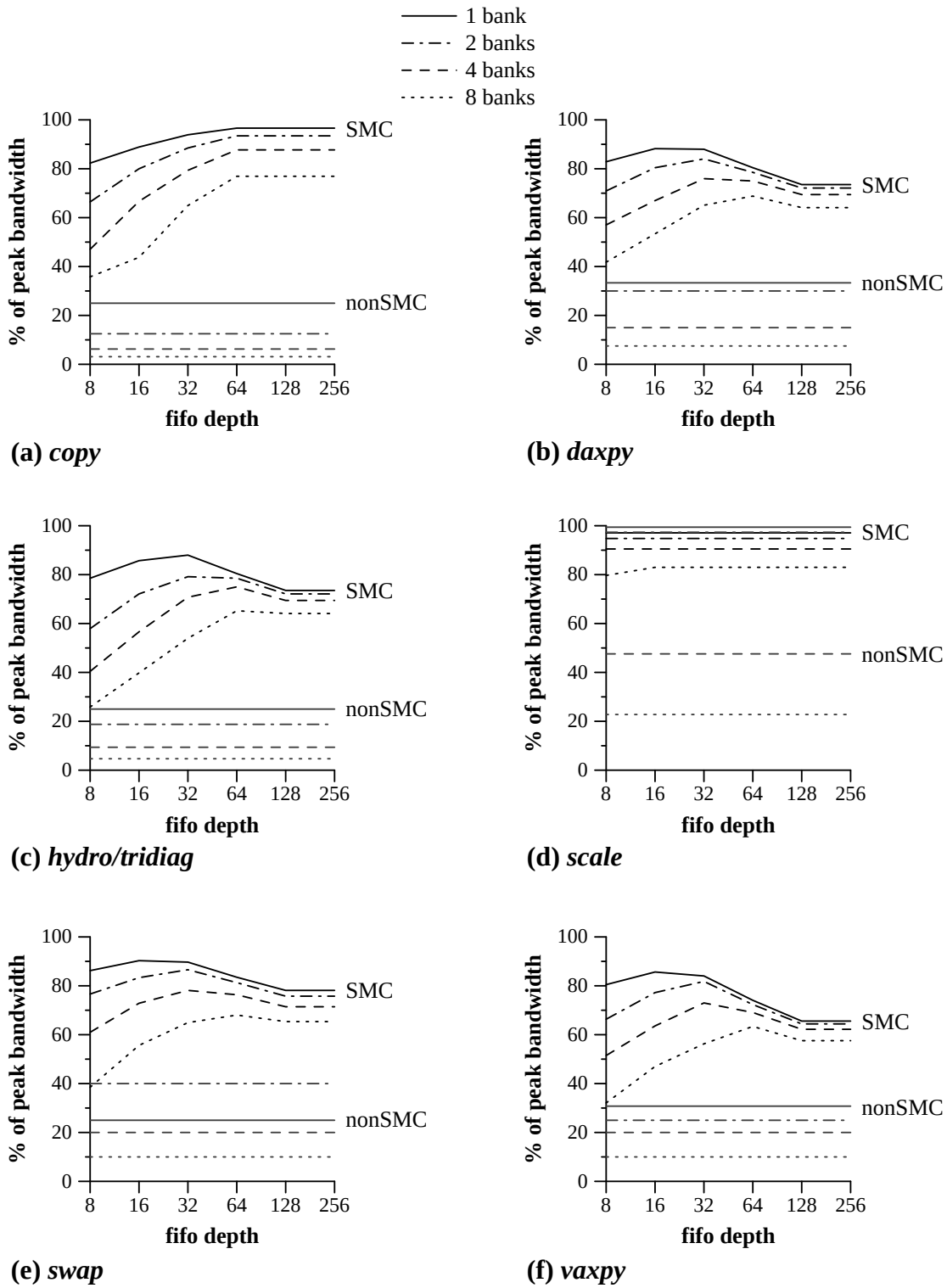


Figure 17 P2 — Medium Vector Performance (for better nonSMC alignment)

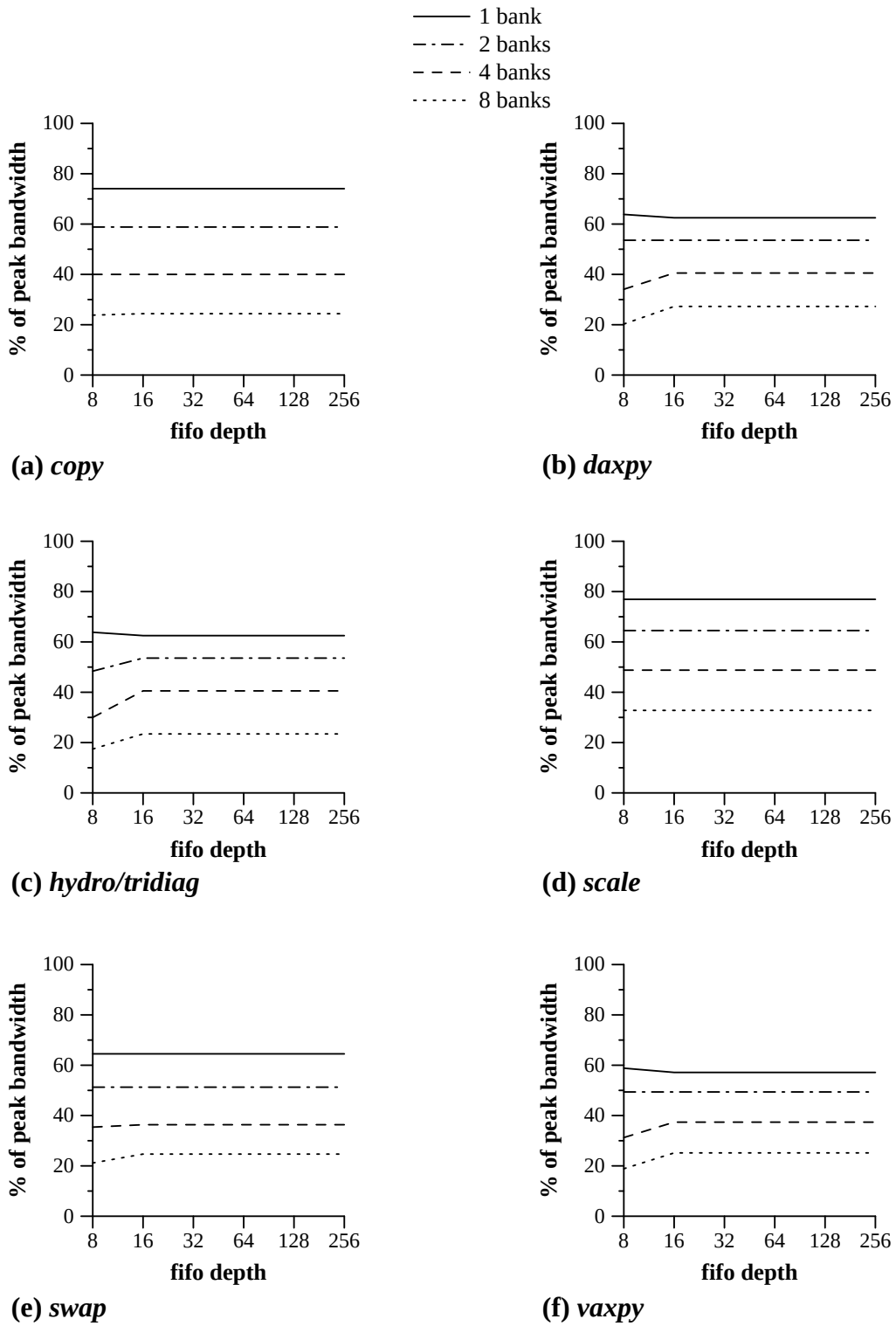


Figure 18 P2 — Short Vector Performance

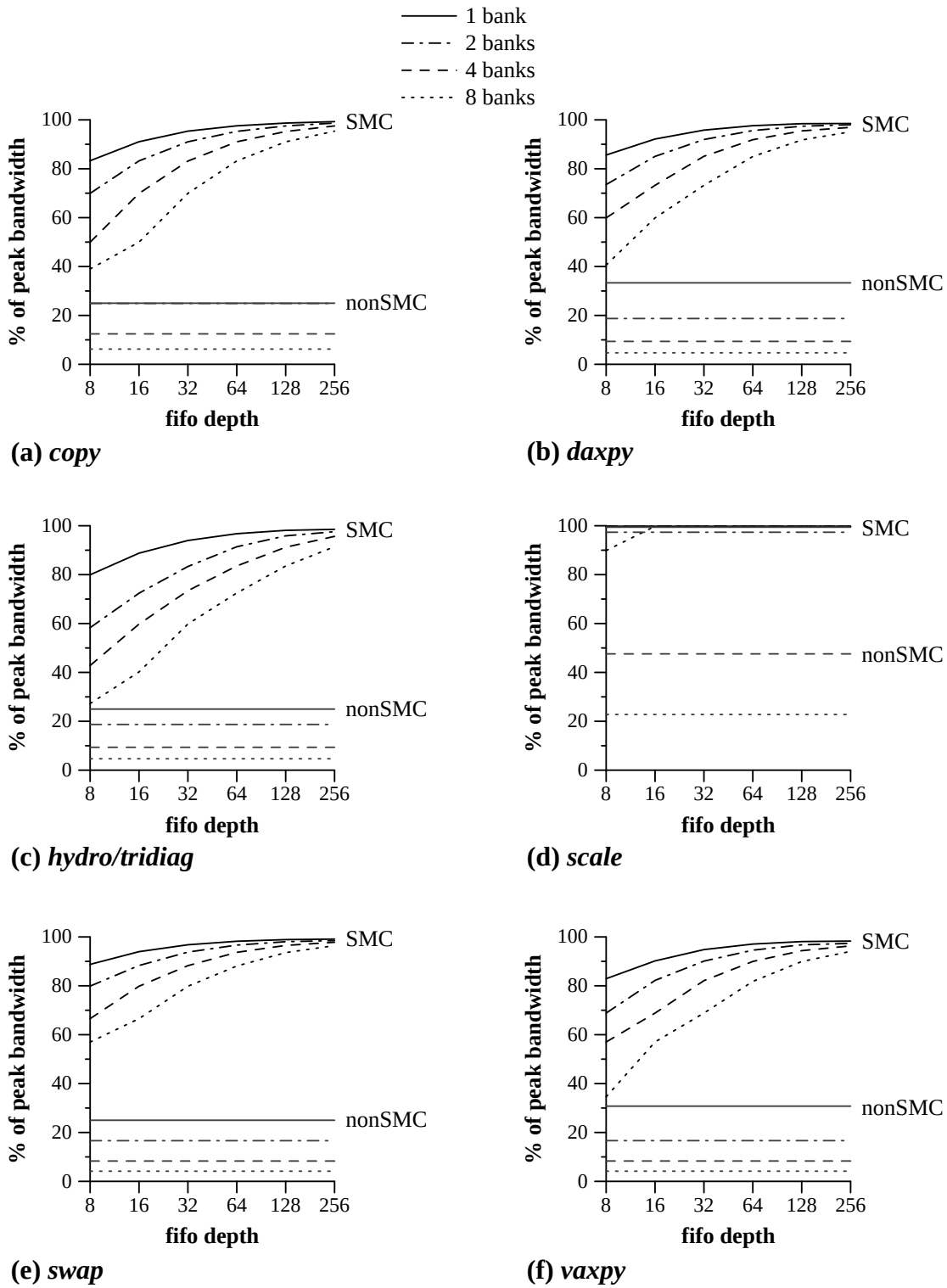


Figure 19 R2 — Long Vector Performance

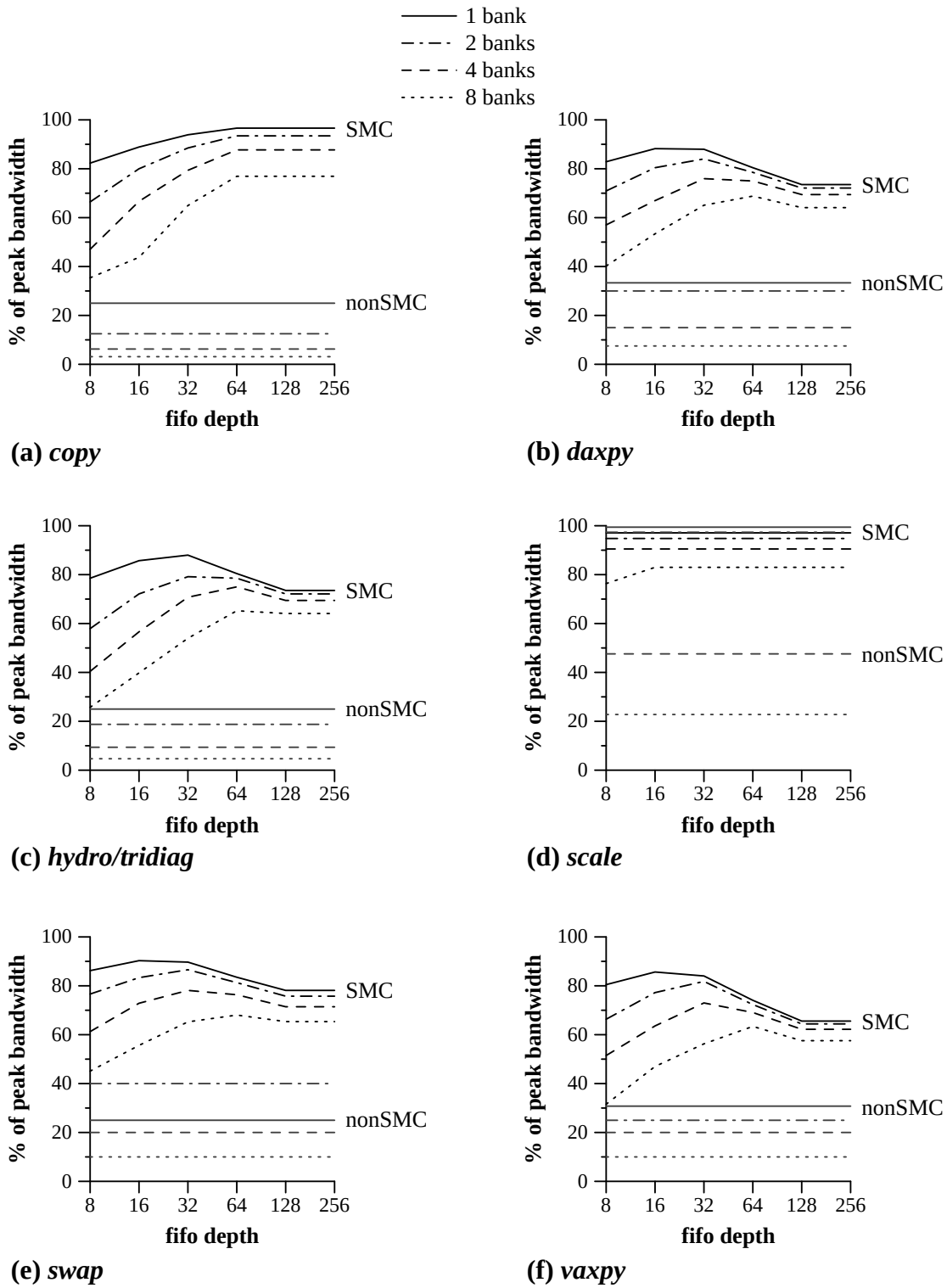


Figure 20 R2 — Medium Vector Performance (for better nonSMC alignment)

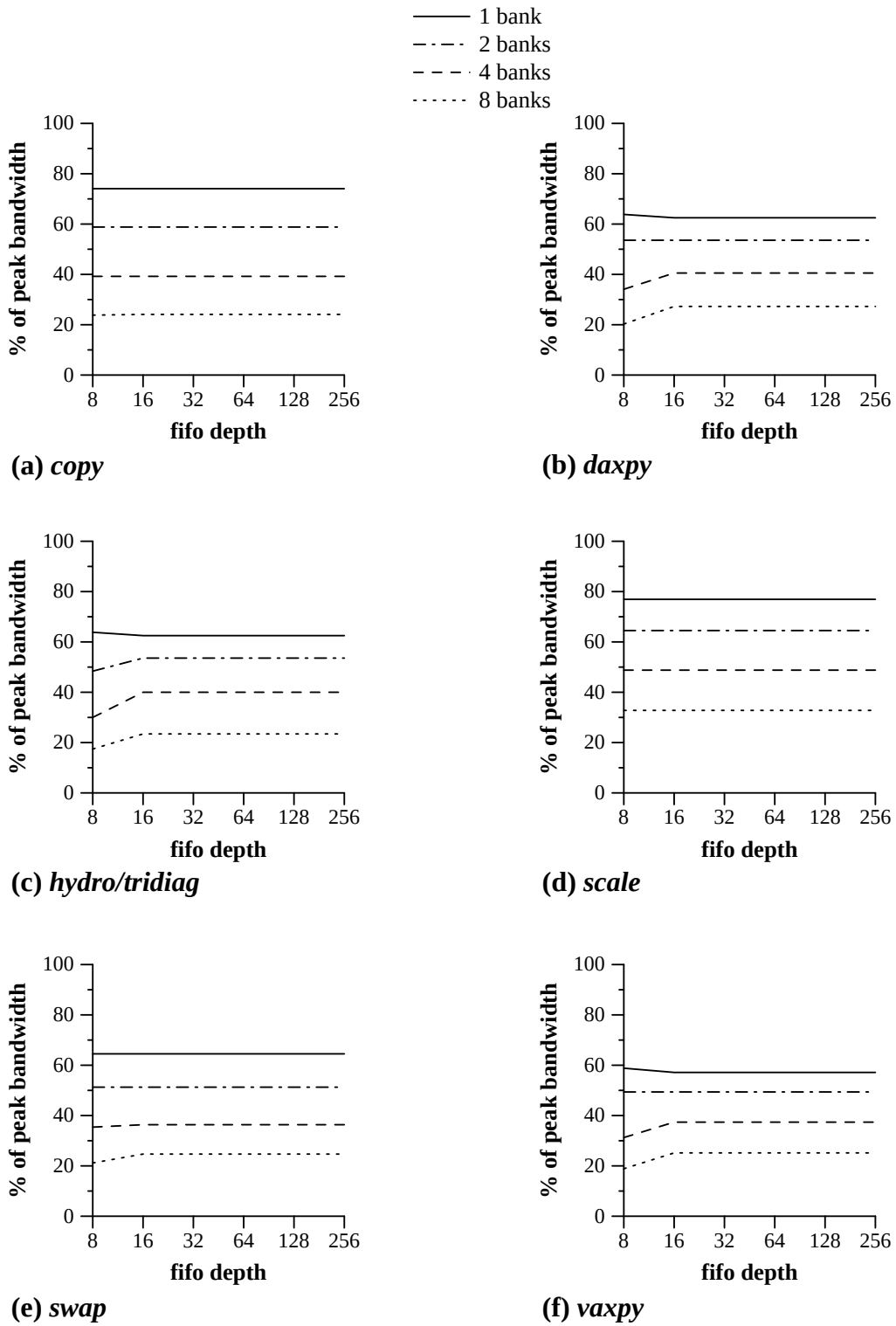


Figure 21 R2 — Short Vector Performance

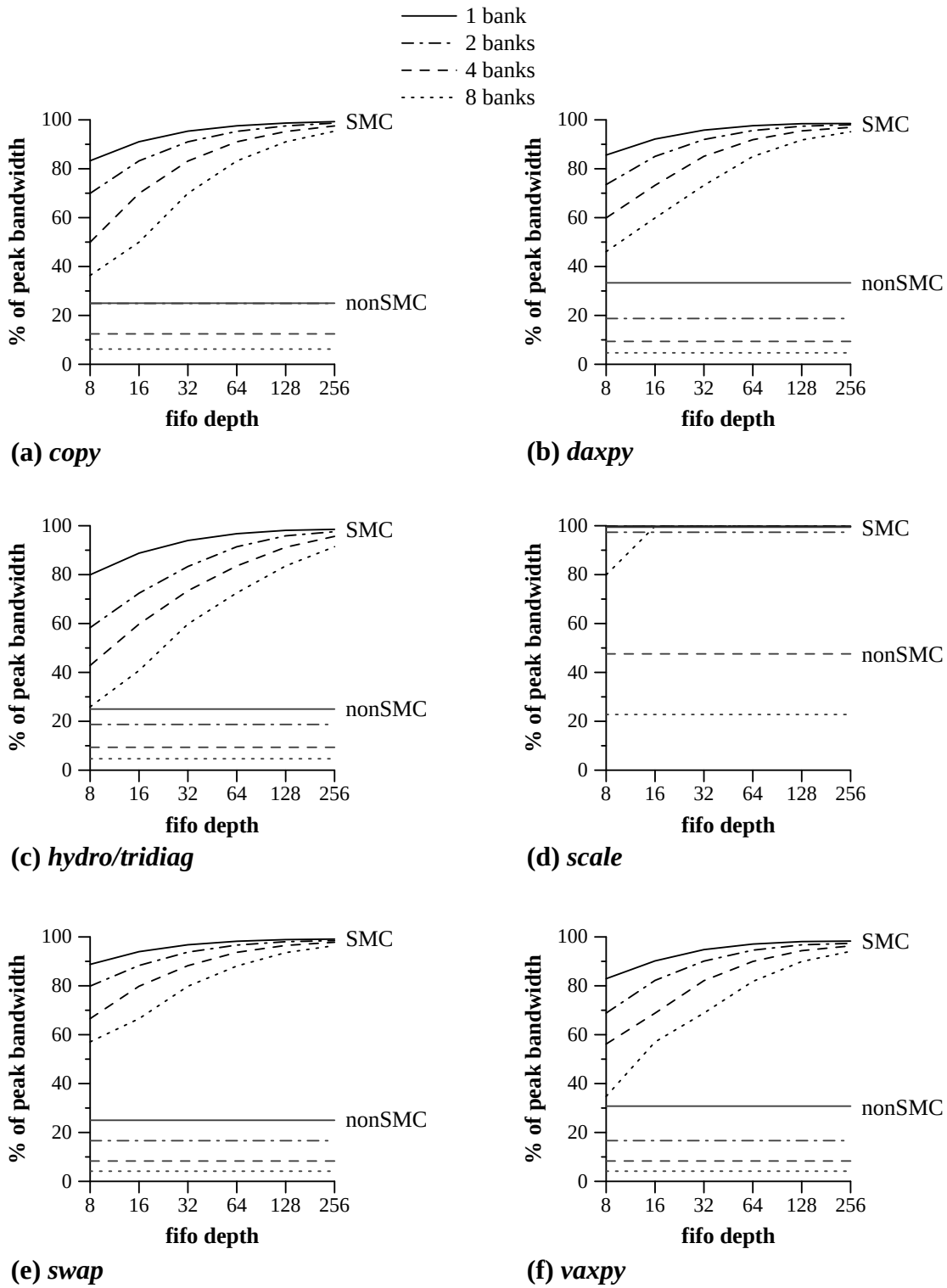


Figure 22 T2 — Long Vector Performance

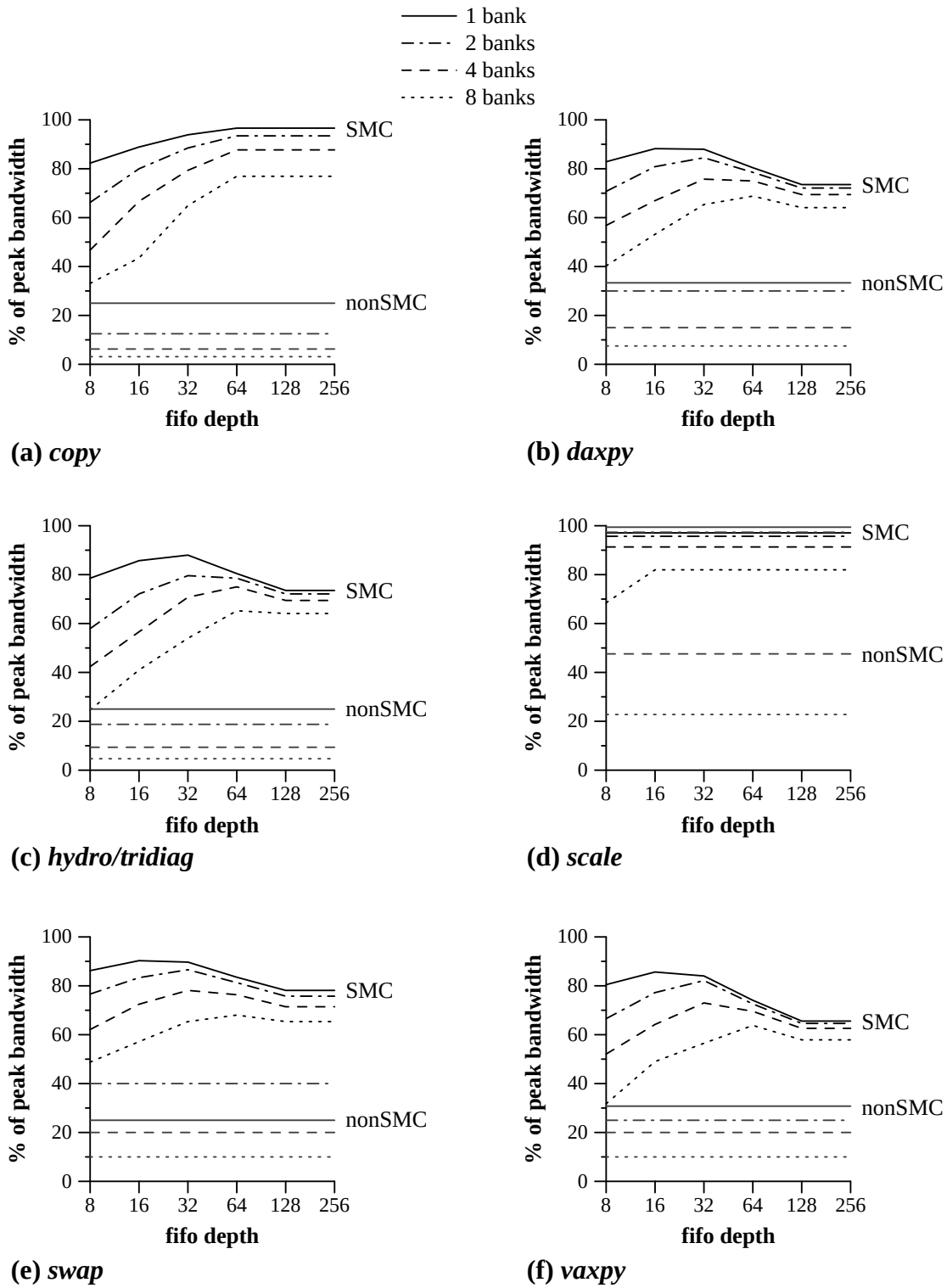


Figure 23 T2 — Medium Vector Performance (for better nonSMC alignment)

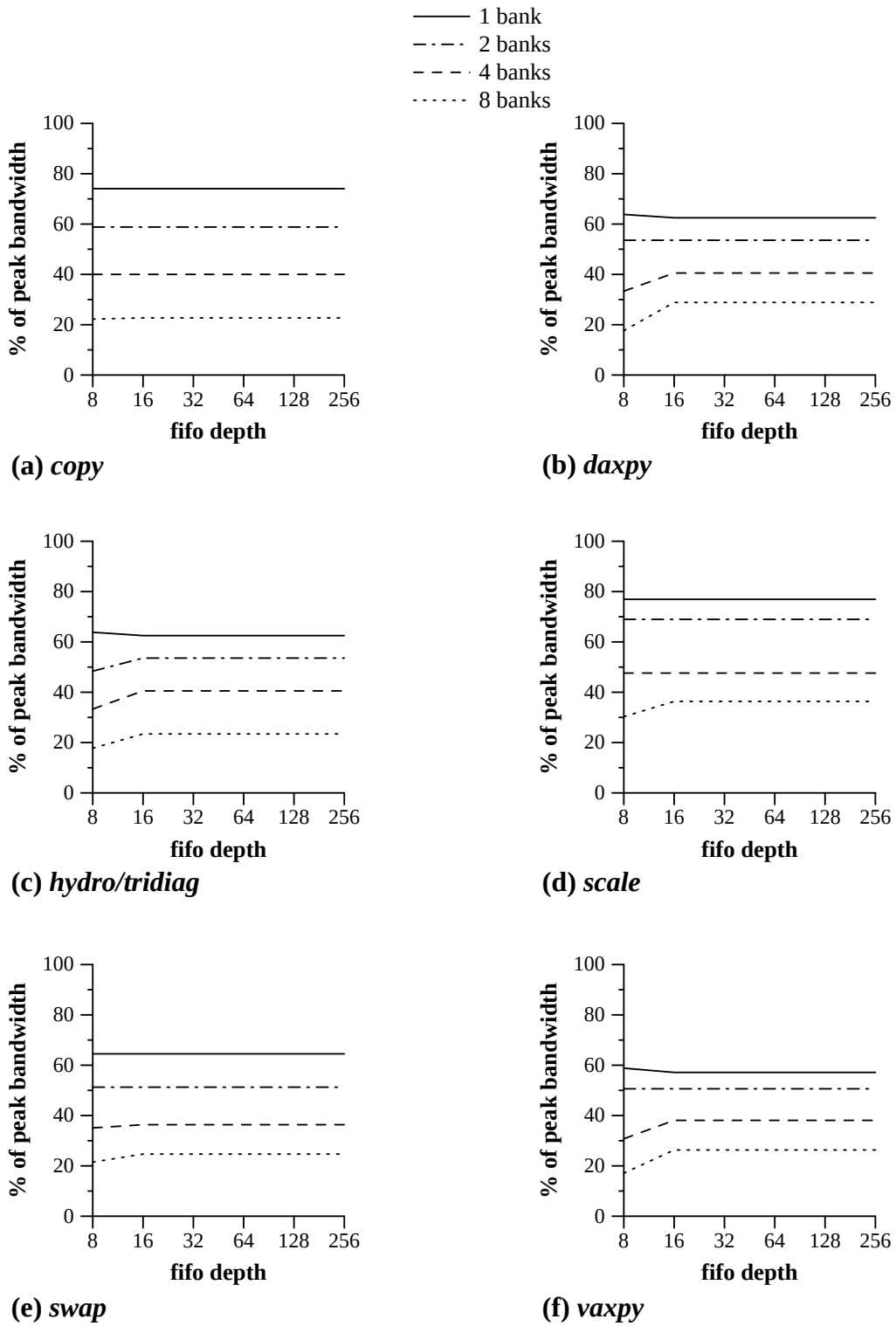


Figure 24 T2 — Short Vector Performance

Table 4 summarizes T2's performance with respect to T1. Recall that numerical values are given only for differences of magnitude greater than or equal to 1%. Values of lesser magnitude are indicated by the sign of the difference. Blank entries indicate that differences, if any, are less than one hundredth of one percent.

Table 4: Performance of Scheme T2 with Respect to T1

Differences in Attained Percentage of Peak Bandwidth												
benchmark	banks	medium vectors					long vectors					
		FIFO depth					FIFO depth					
		8	16	32	64	128	8	16	32	64	128	256
copy	1											
	2								+		+	+
	4									+		+
	8											
daxpy	1							-				
	2						+	-	+	+	+	+
	4							+	+	+	+	-
	8											
hydro	1											
	2	+	-	-2.2			+	-	+	+	+2.2	+1.9
	4		+1.4					+5.9	+1.5	+1.2	+4.1	+4.6
	8				-3.6				+5.9	+6.8	+5.8	+6.6
scale	1											
	2							+	+	+	+	+
	4								+	+	+	+
	8											
swap	1											+
	2						+	-	+	+	+	+
	4							+	+	+	+	+
	8											
vaxpy	1										-	+
	2						+	+	+	+	+	+
	4							+	+	+	+	+
	8											

7.1.3 Group 3 — Algorithms P3, R3, and T3

These algorithms are almost identical to those described in the last section, except that when there are no more ready accesses that hit the current page of the chosen bank and no FIFO meets the required threshold for service, no access is initiated. The intent is to amortize the cost of a DRAM page miss over as many page hits as possible; if we have to switch pages, but there are sufficiently few accesses that would hit the new page, we delay paying the page-miss cost until there are more accesses to offset the overhead.

The performance of the Group 3 algorithms is depicted in Figure 25 through Figure 33. The fact that these algorithms occasionally choose to do nothing has little or no effect on long vector performance, but medium vector performance is tends to be slightly lower than for the algorithms in Group 1 or Group 2, and short vector performance generally suffers a bit more.

For long vectors, the differences in performance between the Group 3 and Group 1 schemes is generally within 1% or 2% of peak bandwidth, plus or minus. Again, the *hydro* benchmark represents the exception. Here the mean performance gain for all FIFO depths and interleaving factors is 4.1% of peak, and the maximum is 10.6% for eight banks and depth-64 FIFOs.

Performance is more erratic for medium vectors, ranging from a 5.8% gain in peak bandwidth to an 11.9% drop (as compared with the corresponding Group 1 algorithms). Performance for short vectors exhibits similar fluctuations, ranging from a 6.9% increase in attainable bandwidth for the *daxpy* and *hydro* benchmarks, to a 15.3% decrease for *copy.*

Again, there is no discernible pattern to the performance variations, but now *scale* is the only benchmark whose performance remains unchanged. For instance, Algorithm R3's performance on *daxpy* for 100-element vectors and a four-bank memory using sixteen-deep FIFOs is 67.0% of peak. R2 and R1 both deliver 69.3%, a difference of only a few percent.

On the *copy* benchmark on a two-bank system with eight-deep FIFOs, however, the difference goes the other way — R3 attains 68.3% of peak, whereas R2 and R1 deliver 66.4%. For FIFOs of sixteen double-words and the same number of banks, R2 and R1 once again win out with 80.0% over 77.8%.

There seems to be little advantage in waiting for a certain number of accesses to a DRAM page to accumulate before paying the page-miss overhead — doing so occasionally improves bandwidth, but it also frequently diminishes performance, and the drops we've seen are about twice as large as the gains. Indeed, performance may suffer appreciably under such a policy. This is good news from an implementation standpoint, since incorporating the threshold would require extra circuitry, and would complicate the selection logic.

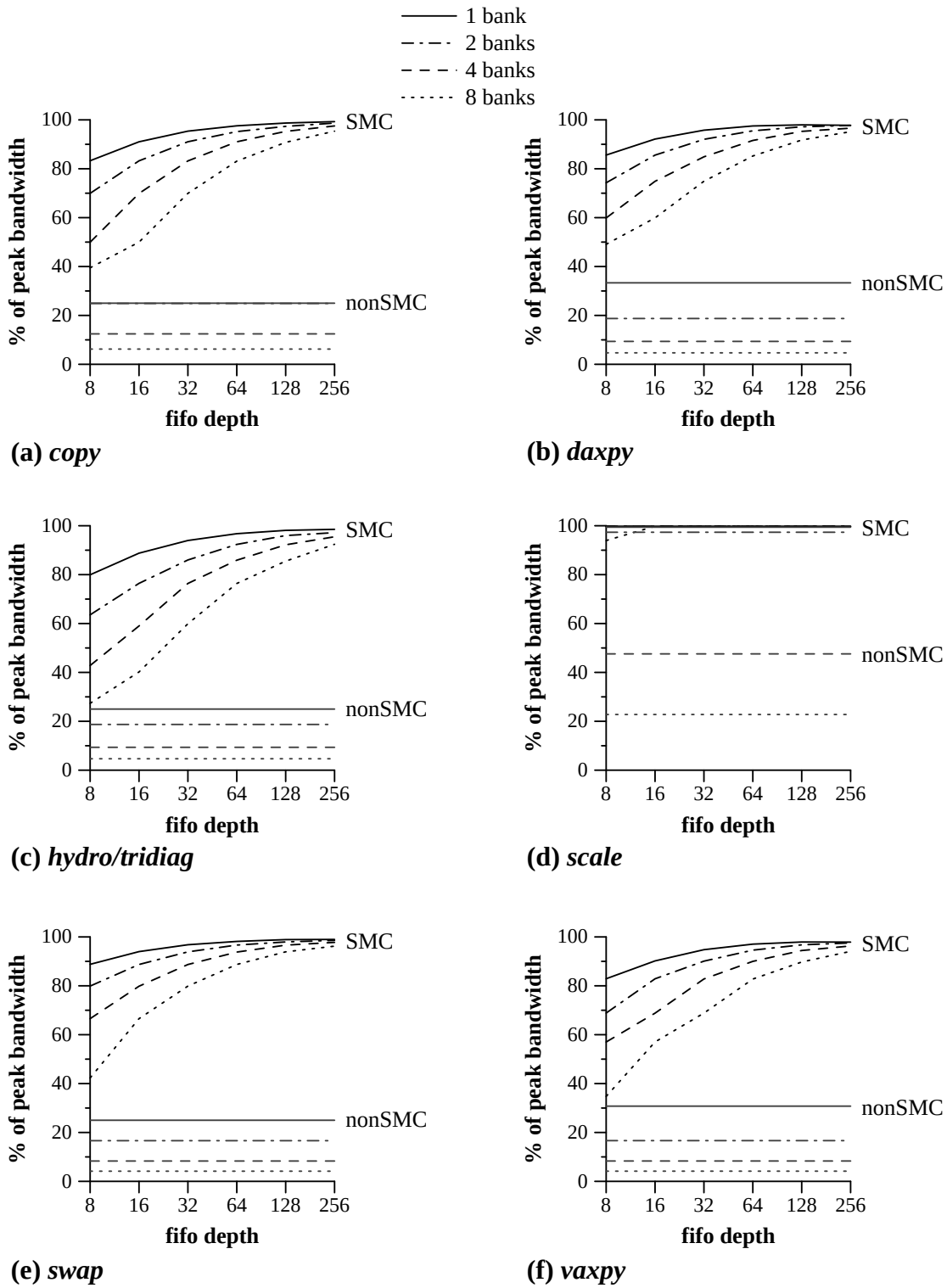


Figure 25 P3 — Long Vector Performance

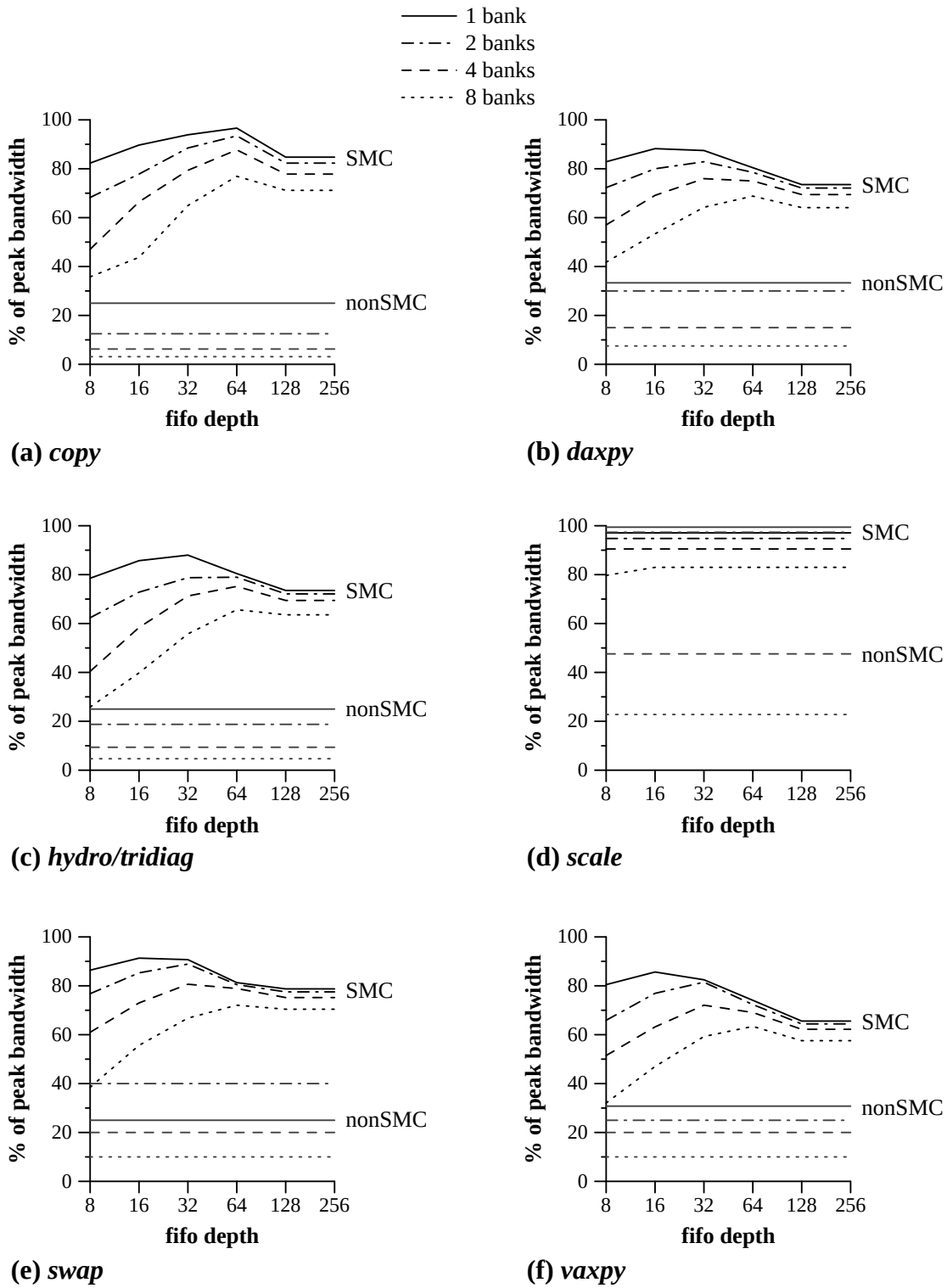


Figure 26 P3 — Medium Vector Performance (for better nonSMC alignment)

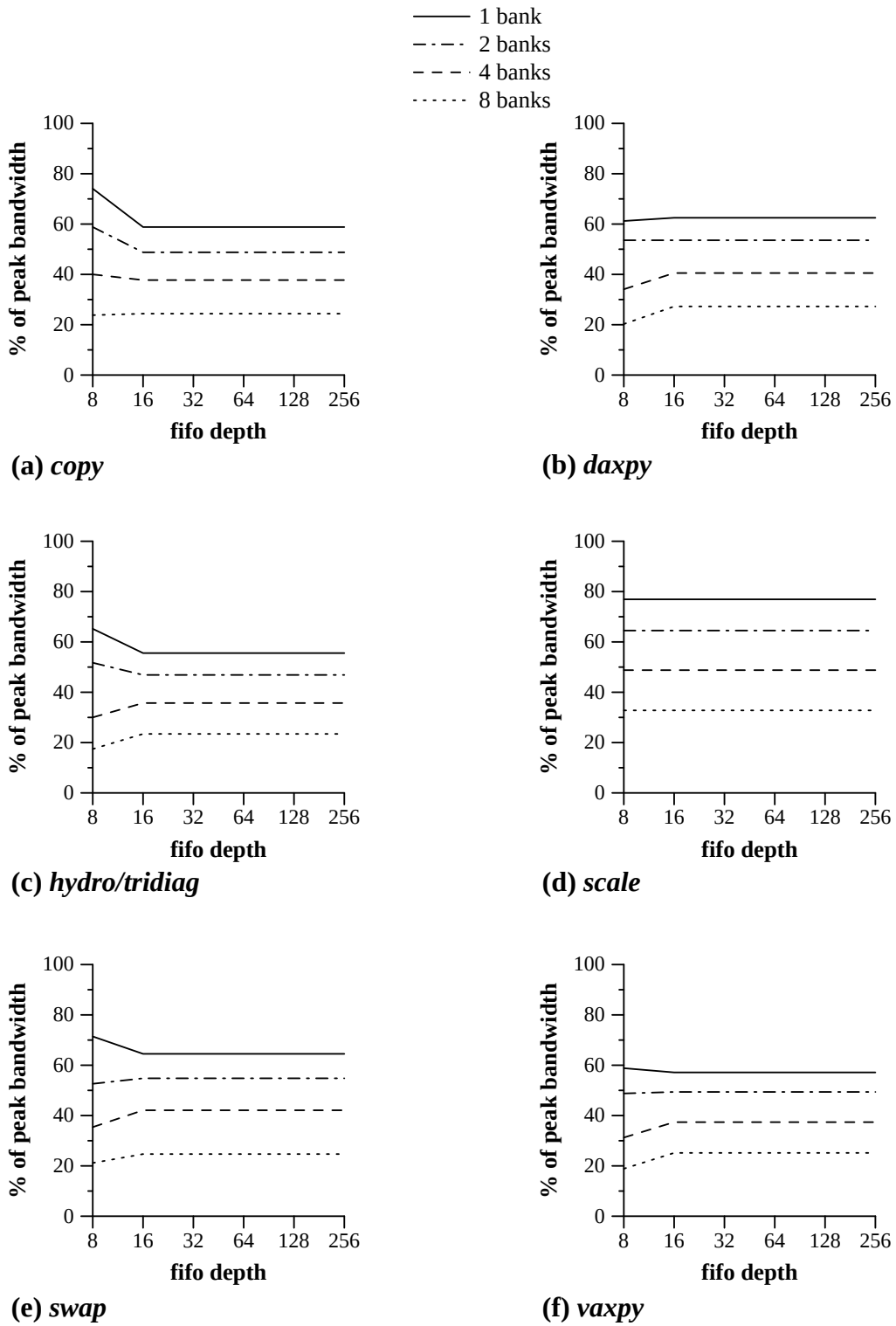


Figure 27 P3 — Short Vector Performance

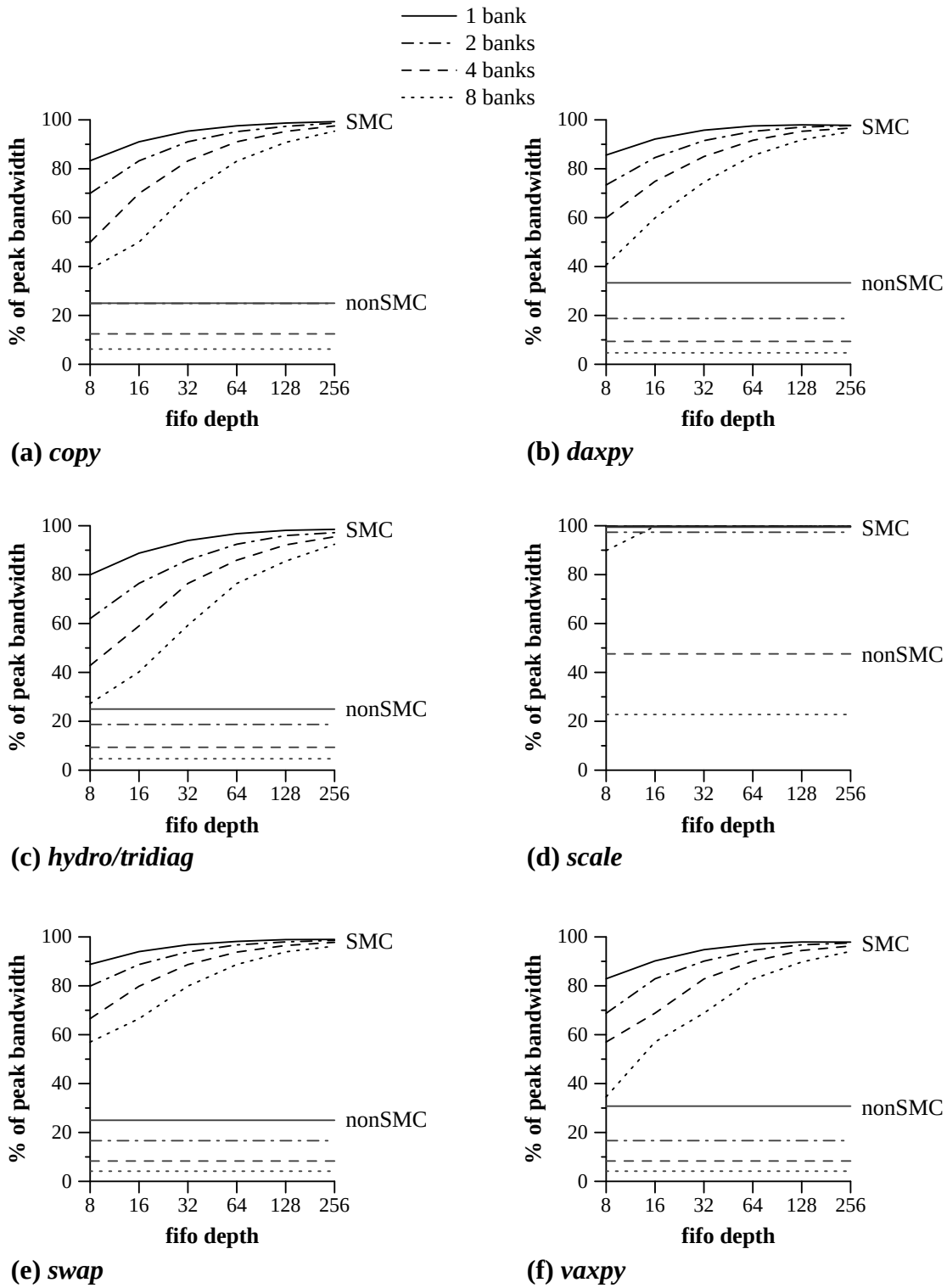


Figure 28 R3 — Long Vector Performance

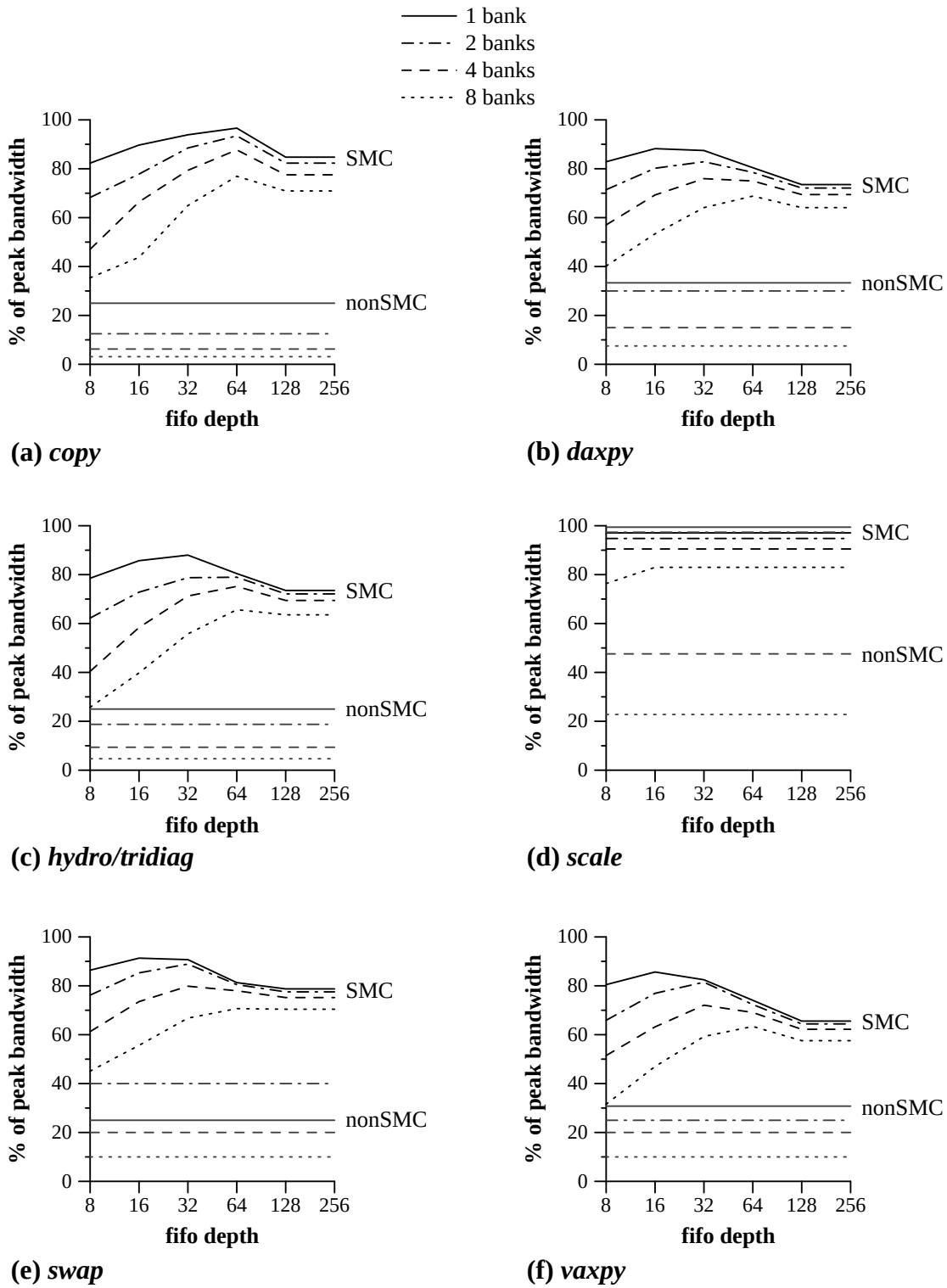


Figure 29 R3 — Medium Vector Performance (for better nonSMC alignment)

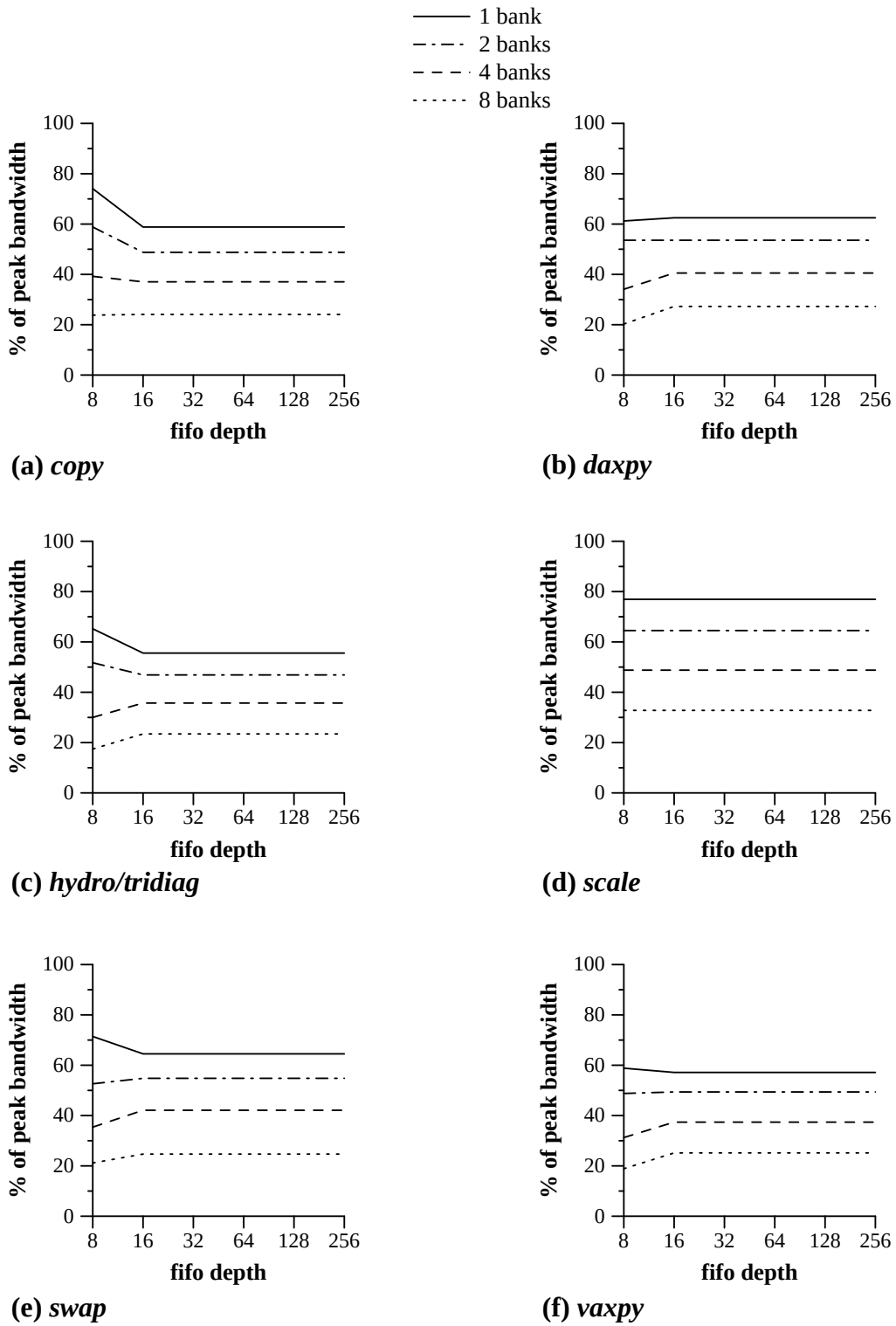


Figure 30 R3 — Short Vector Performance

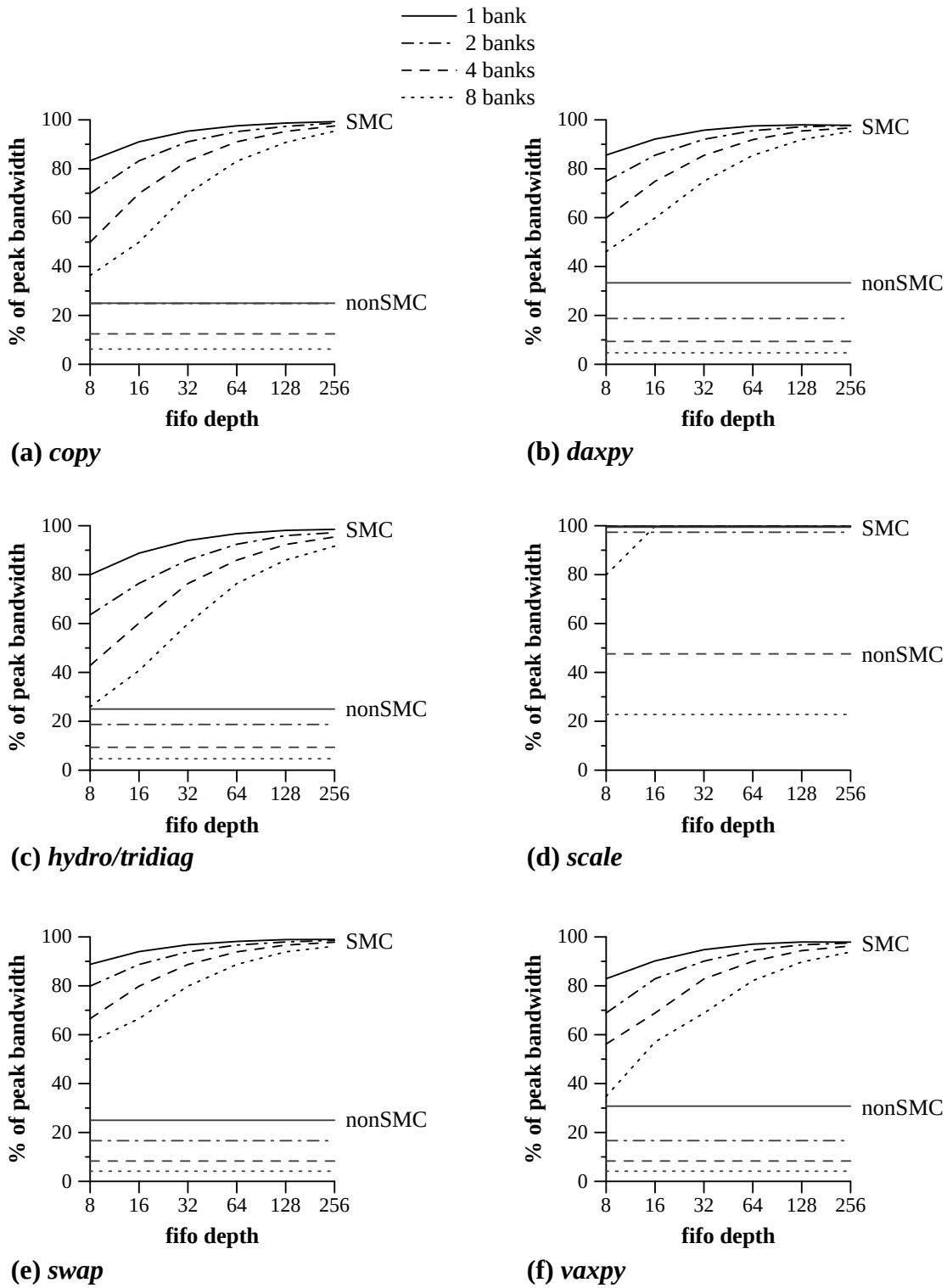


Figure 31 T3 — Long Vector Performance

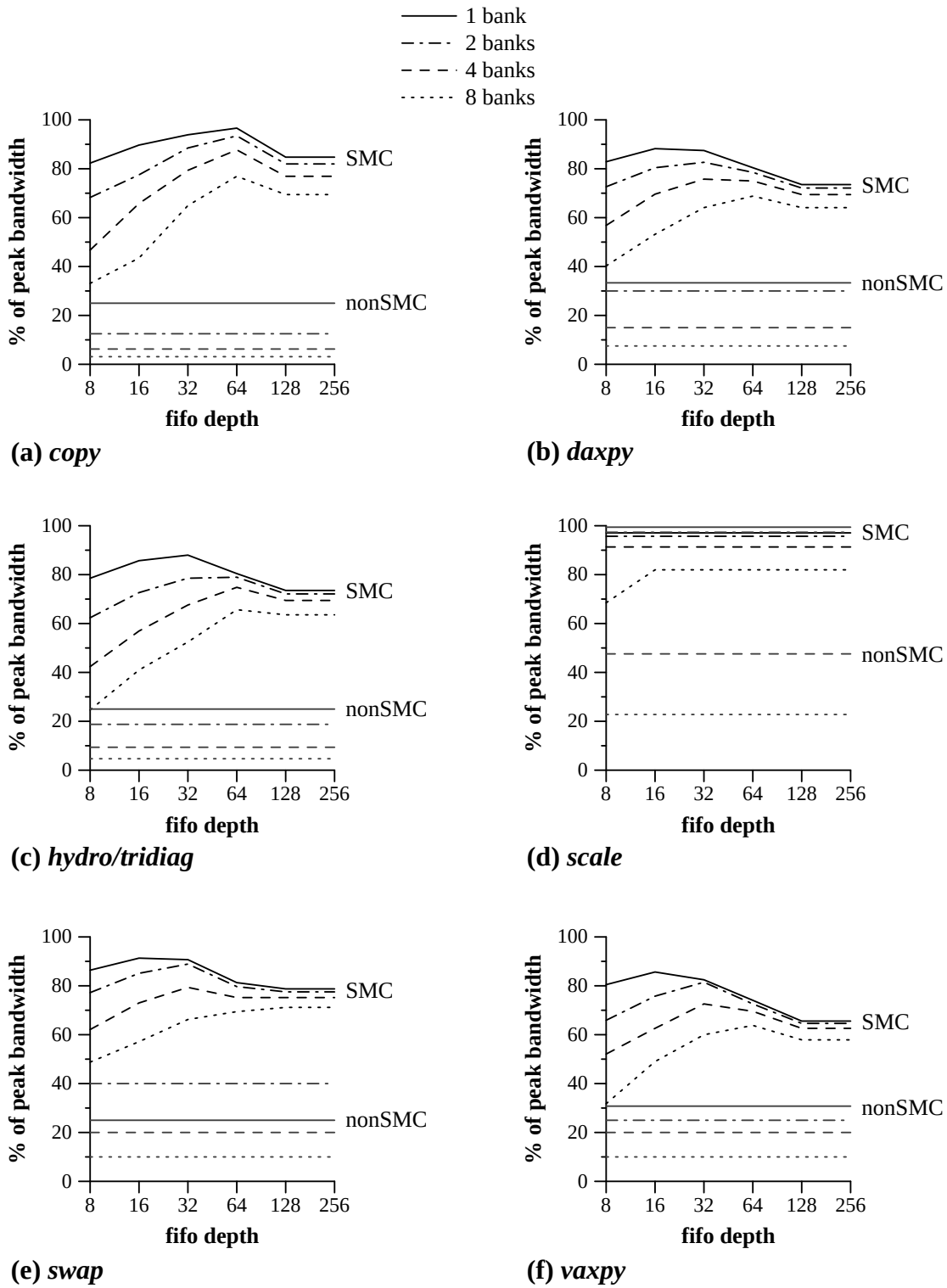


Figure 32 T3 — Medium Vector Performance (for better nonSMC alignment)

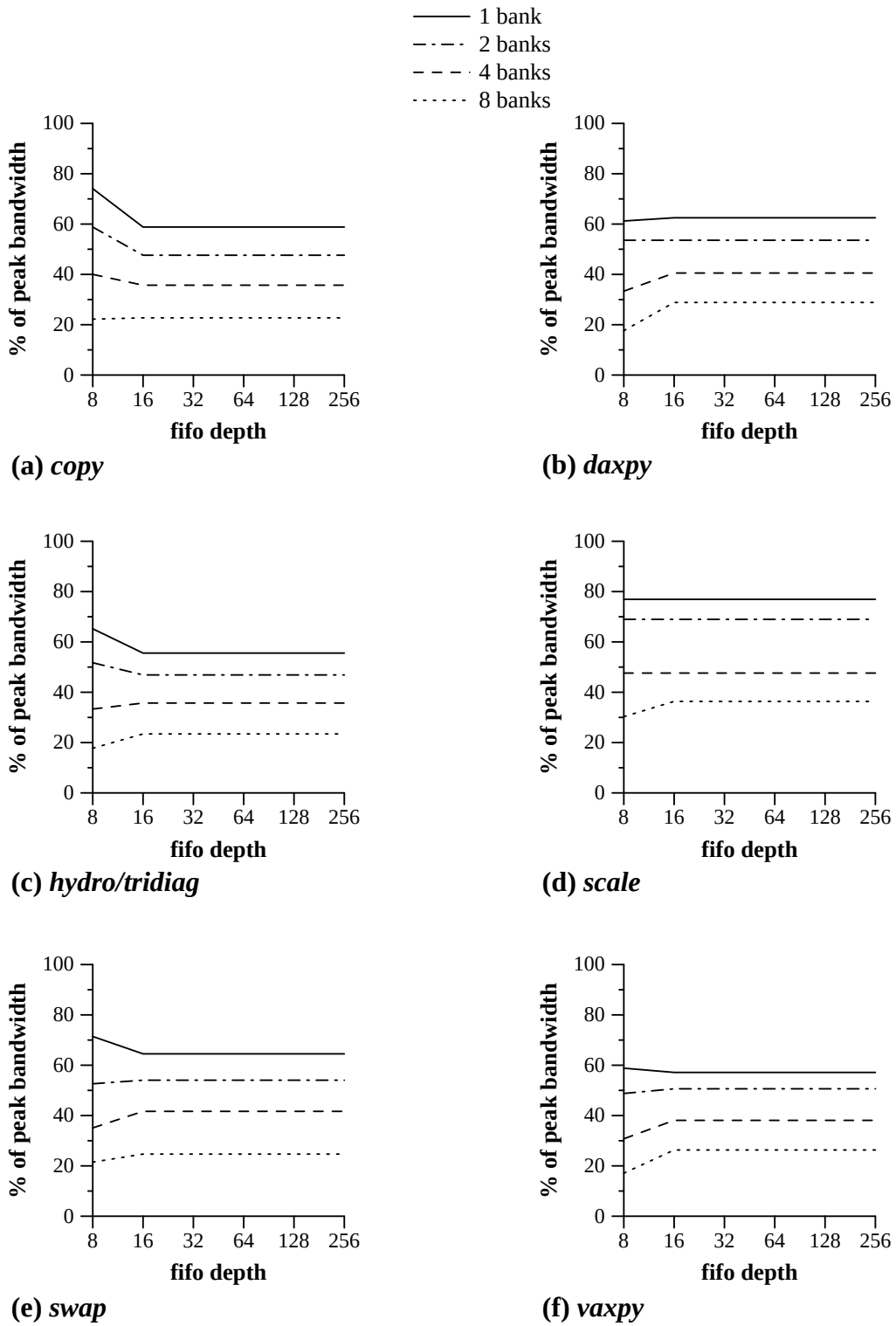


Figure 33 T3 — Short Vector Performance

Table 5 summarizes T3's performance with respect to T1. Recall that blank entries indicate that differences, if any, are less than 0.01%. Numerical values are given for differences of magnitude greater than 1%; entries of lesser magnitude are represented by the sign of the difference.

Table 5: Performance of Scheme T3 with Respect to T1

Differences in Attained Percentage of Peak Bandwidth												
benchmark	banks	medium vectors					long vectors					
		FIFO depth					FIFO depth					
		8	16	32	64	128	8	16	32	64	128	256
copy	1		+			-11.9		-			+	
	2	+2.0	-2.5			-11.5		+	+	-	-	+
	4		-			-10.8		-	+	+	+	+
	8					-7.5			-	-	-	
daxpy	1			-			-	-	-	-	-	-
	2	+1.9	-	-1.9			+1.4	+	+		-	-
	4							+1.7	+	+	+	-
	8			-1.3					+1.7	+	+	+
hydro	1											
	2	+5.0	+	-3.2	+		+6.0	+3.8	+2.7	+1.9	+2.3	+1.5
	4		+1.8	-3.2	-			+6.3	+4.4	+3.6	+5.2	+4.4
	8			-1.5	-3.2	-			+6.0	+10.6	+8.2	+6.8
scale	1											
	2							+	+	+	+	+
	4								+	+	+	+
	8											
swap	1	+	+1.0	+1.0	-2.2	+	+	+	+	-		-
	2	+	+1.8	+2.3	-1.6	+1.8	+	+	+	+	+	+
	4		+	+1.2	-1.2	+3.8		+	+	+	+	+
	8			+	+1.4	+5.8			+	+	+	-
vaxpy	1			-1.6			-		-	-	-	-
	2	-	-1.5	-			+	+	+	+	+	+
	4		-1.6	-				+	+	+	+	+
	8			+3.5					-	+	-	-

7.1.4 Group 4 — Algorithms P4, R4, and T4

These algorithms simply look for accesses that hit the current page of the selected bank, and if they find none, they choose the next FIFO in sequence. Unlike the previous schemes, they do not try to choose the "best" FIFO to service in the event of a necessary page miss.

Intuitively, it would seem that these "less intelligent" algorithms would not perform as well as their more sophisticated counterparts in Groups 1 through 3. This turns out not to be the case. As depicted in Figure 34 through Figure 42, performance of these algorithms rivals that of the corresponding members of Group 1 and Group 2.

For long vectors, shown in Figure 34, Figure 37, and Figure 40, percentages of peak bandwidth obtained by these algorithms are usually within a few tenths (plus *or* minus) of those obtained by the more sophisticated algorithms. For the *hydro* benchmark, these algorithms often beat the others by over 10% of the attainable bandwidth (up to 13.2%, in the case of depth-64 FIFOs on an eight-bank memory system).

For medium vectors, depicted in Figure 35, Figure 38, and Figure 41, performance is virtually identical to that for Group 1 on most benchmarks (*copy, daxpy, scale*, and *vaxpy*). *Hydro* again benefits from this simpler FIFO-selection algorithm, although by a somewhat smaller margin than for long vectors. For eight- and sixteen-deep FIFOs, T4 delivers 62.3% and 76.5% of peak bandwidth on a two-bank system, whereas T1 reaches only 57.4% and 72.3% — a difference of over 4% of peak in both cases. On an eight-bank memory using a FIFO depth of sixty-four, however, T4 delivers only 65.2% of the attainable bandwidth, but T1 is able to deliver 68.8%. T1 again beats T4 by a few percent on the *swap* benchmark for very shallow FIFOs on two- and eight-bank systems. Performance for the P and R schemes is similar: *hydro* performance of the Group 4 schemes is several percent better than that of the corresponding Group 1 schemes in some cases, but *swap* performance tends to be a few percent worse in others.

The short vector performance shown in Figure 36, Figure 39, and Figure 42 is precisely the same as for Group 1, except for *hydro*. Here the Group 4 schemes deliver slightly over 5% less of peak bandwidth than the Group 1 schemes for very shallow FIFOs and a two-bank memory, and they exhibit smaller performance fluctuations for memory systems with a higher interleaving factor.

This set of algorithms both performs well (for deeper FIFOs, performance is very competitive with that of the corresponding Group 1 schemes) and would be easier to implement than the others described thus far. The combination of bank-selection and FIFO-selection algorithms represented by T4 would be particularly straightforward.

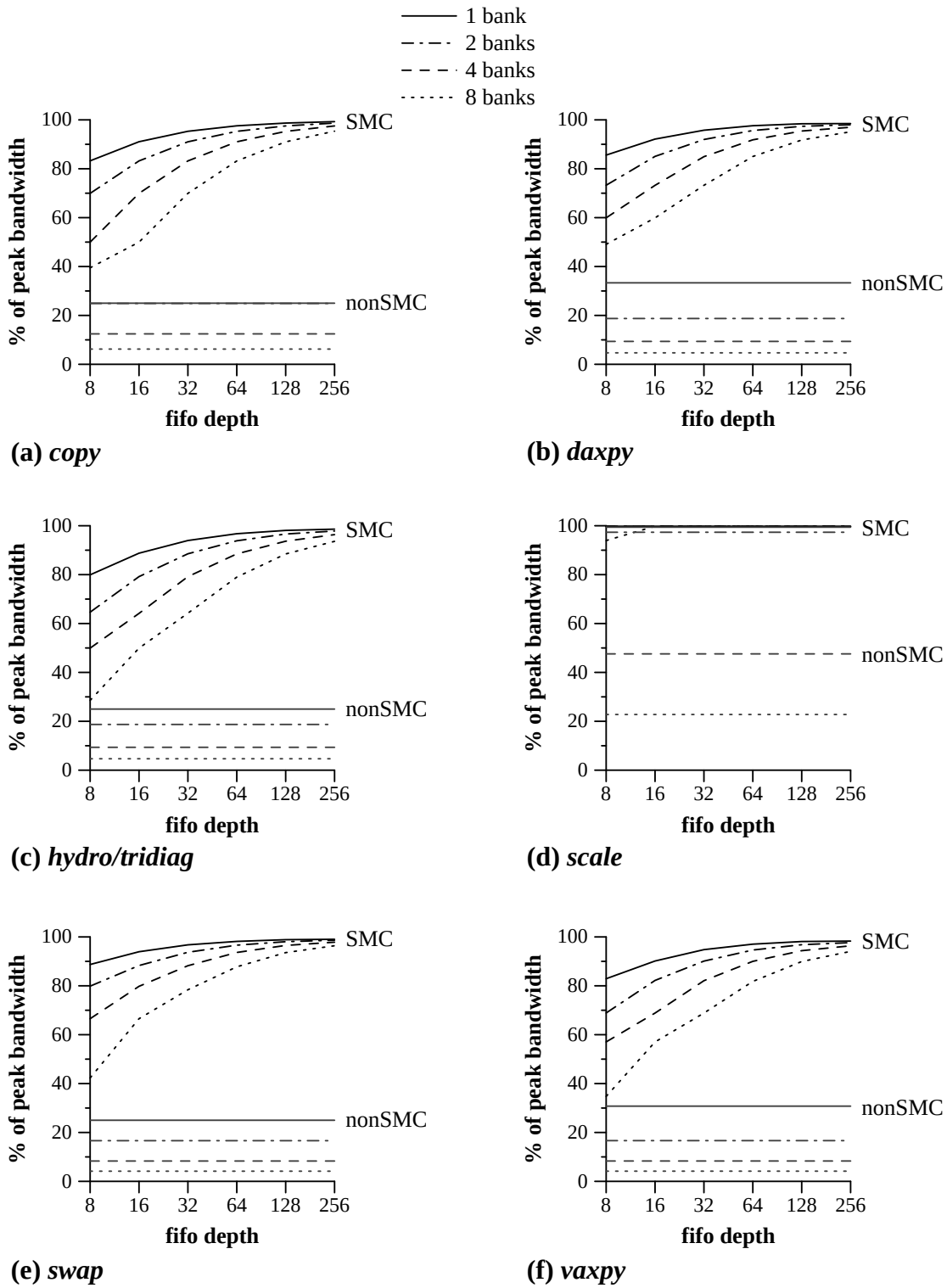


Figure 34 P4 — Long Vector Performance

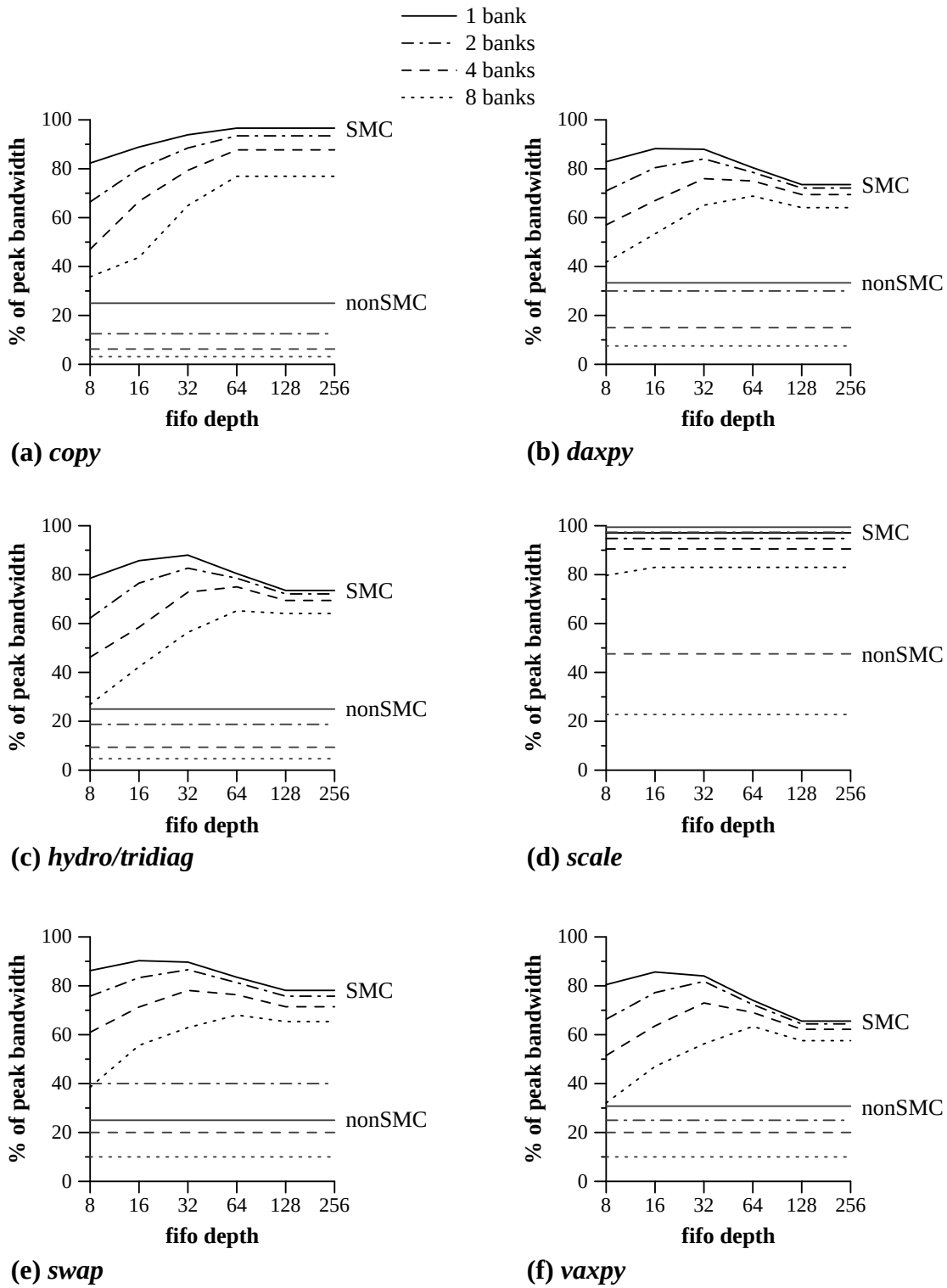


Figure 35 P4 — Medium Vector Performance (for better nonSMC alignment)

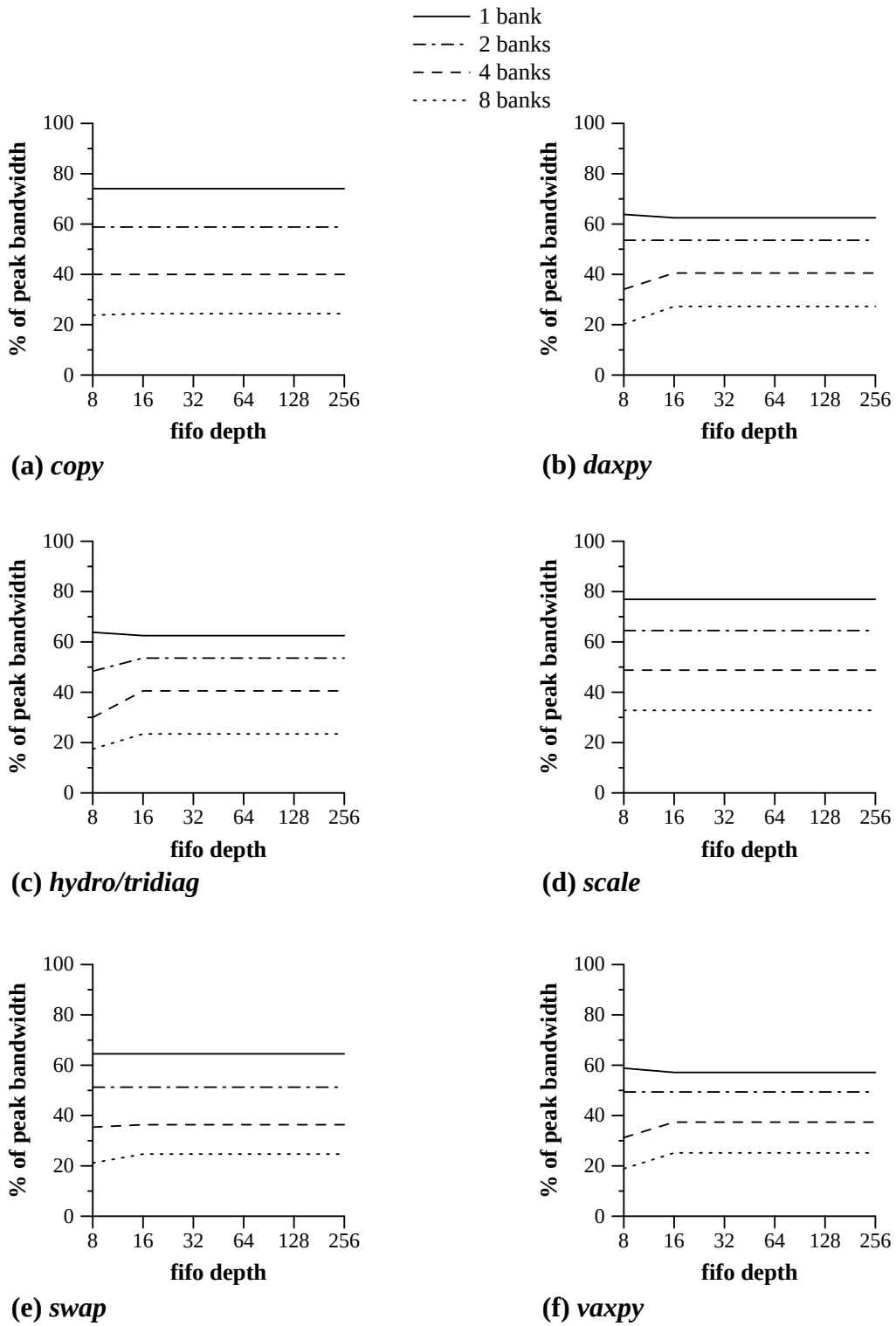


Figure 36 P4 — Short Vector Performance

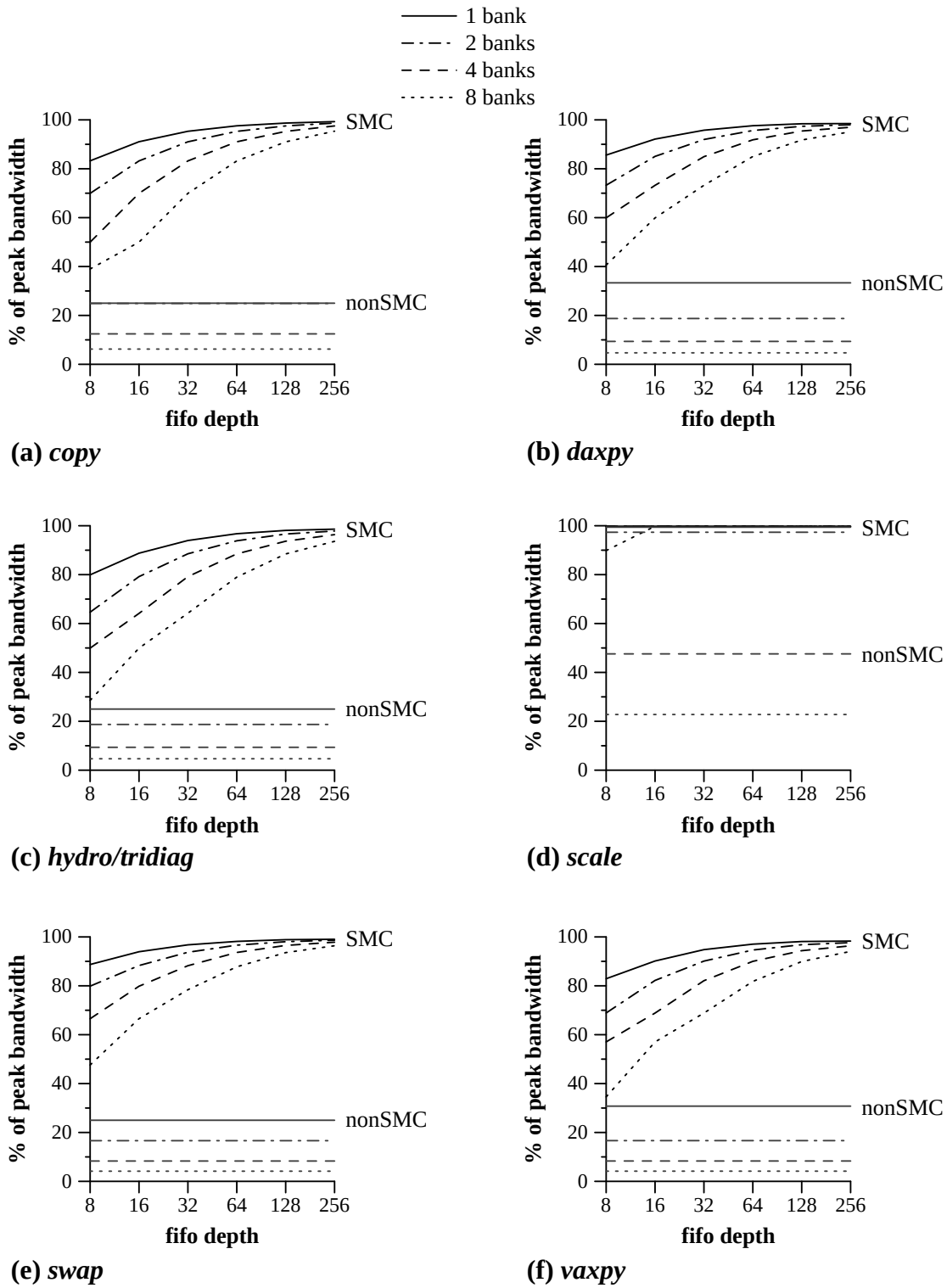


Figure 37 R4 — Long Vector Performance

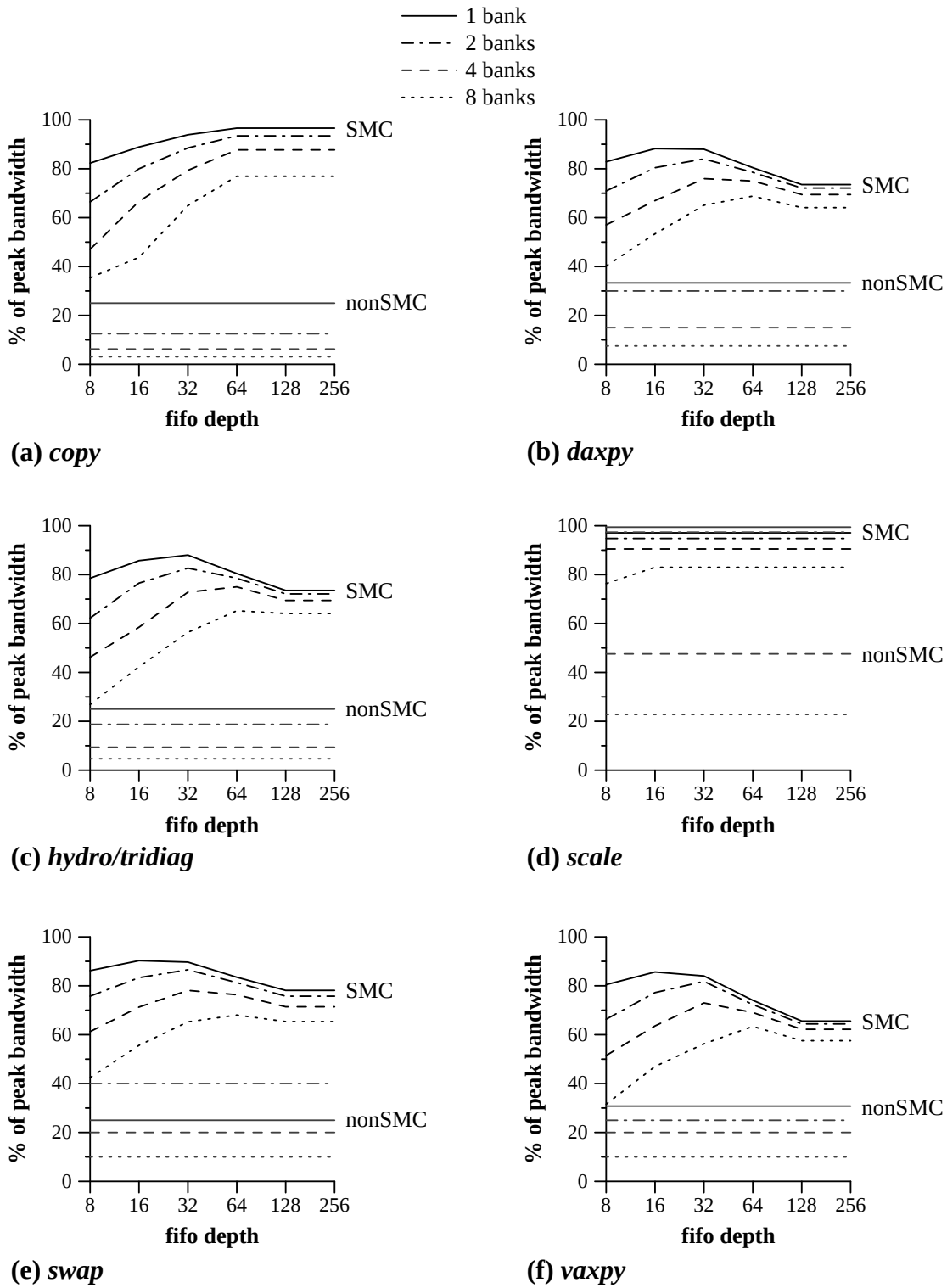


Figure 38 R4 — Medium Vector Performance (for better nonSMC alignment)

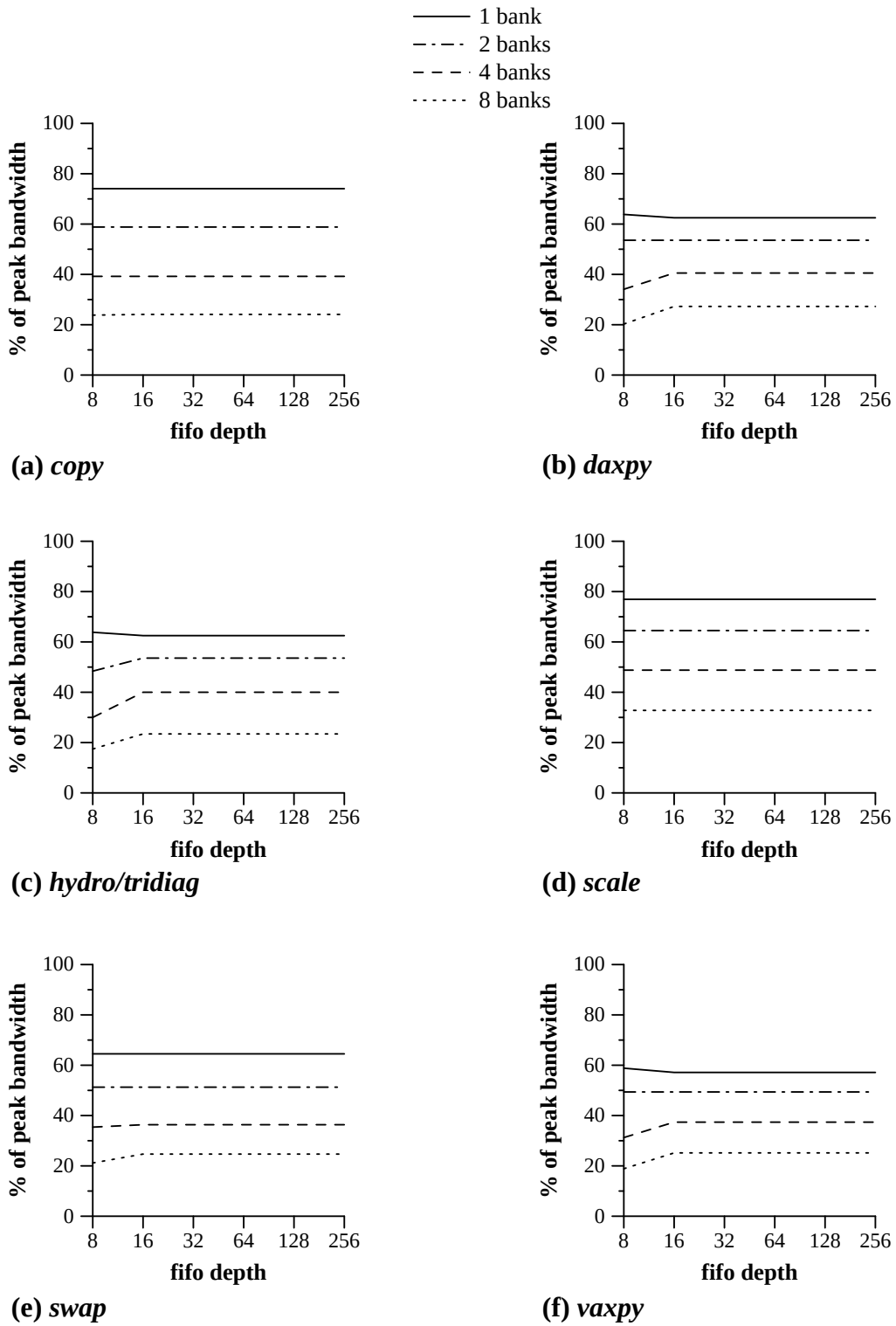


Figure 39 R4 — Short Vector Performance

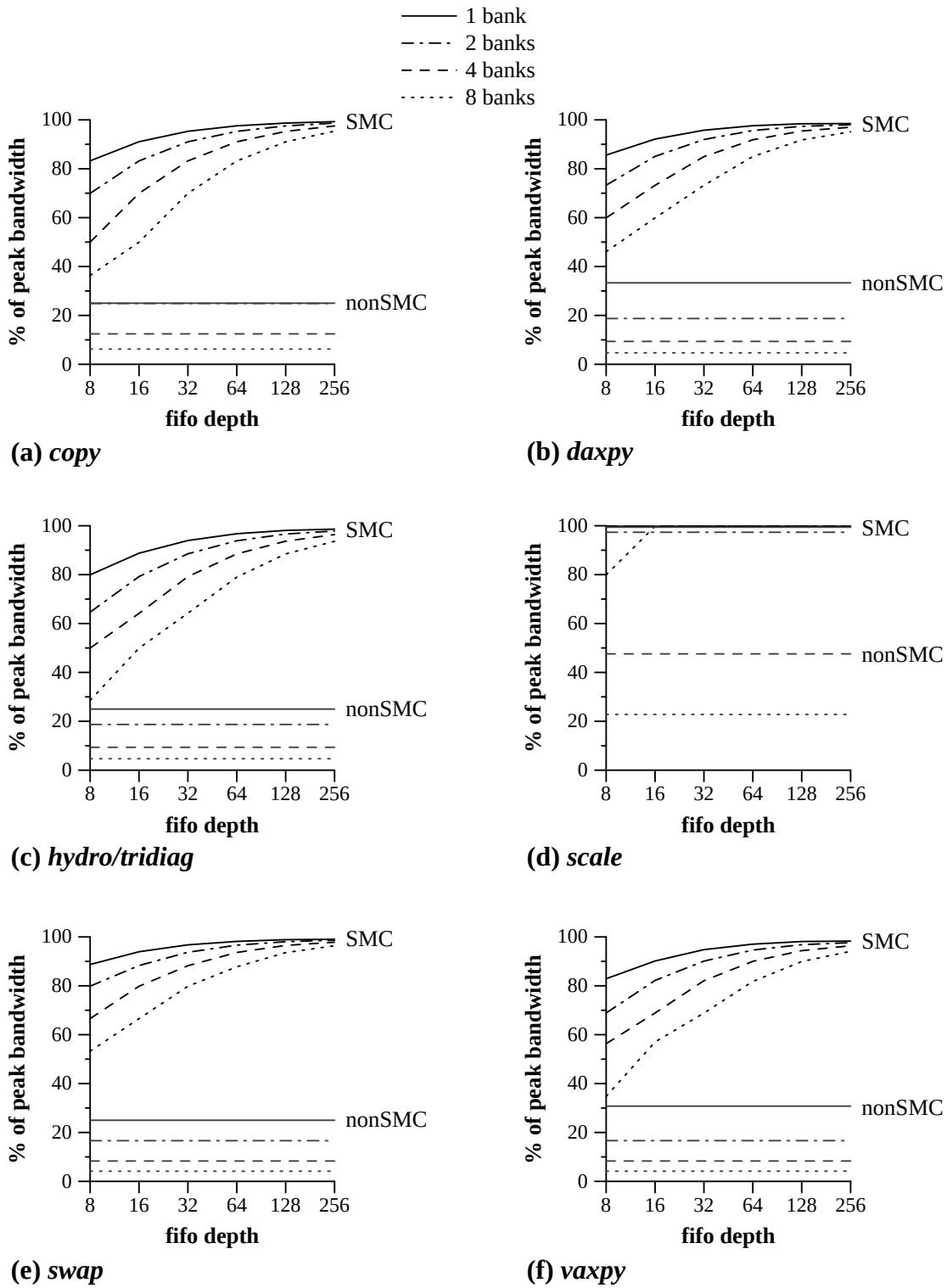


Figure 40 T4 — Long Vector Performance

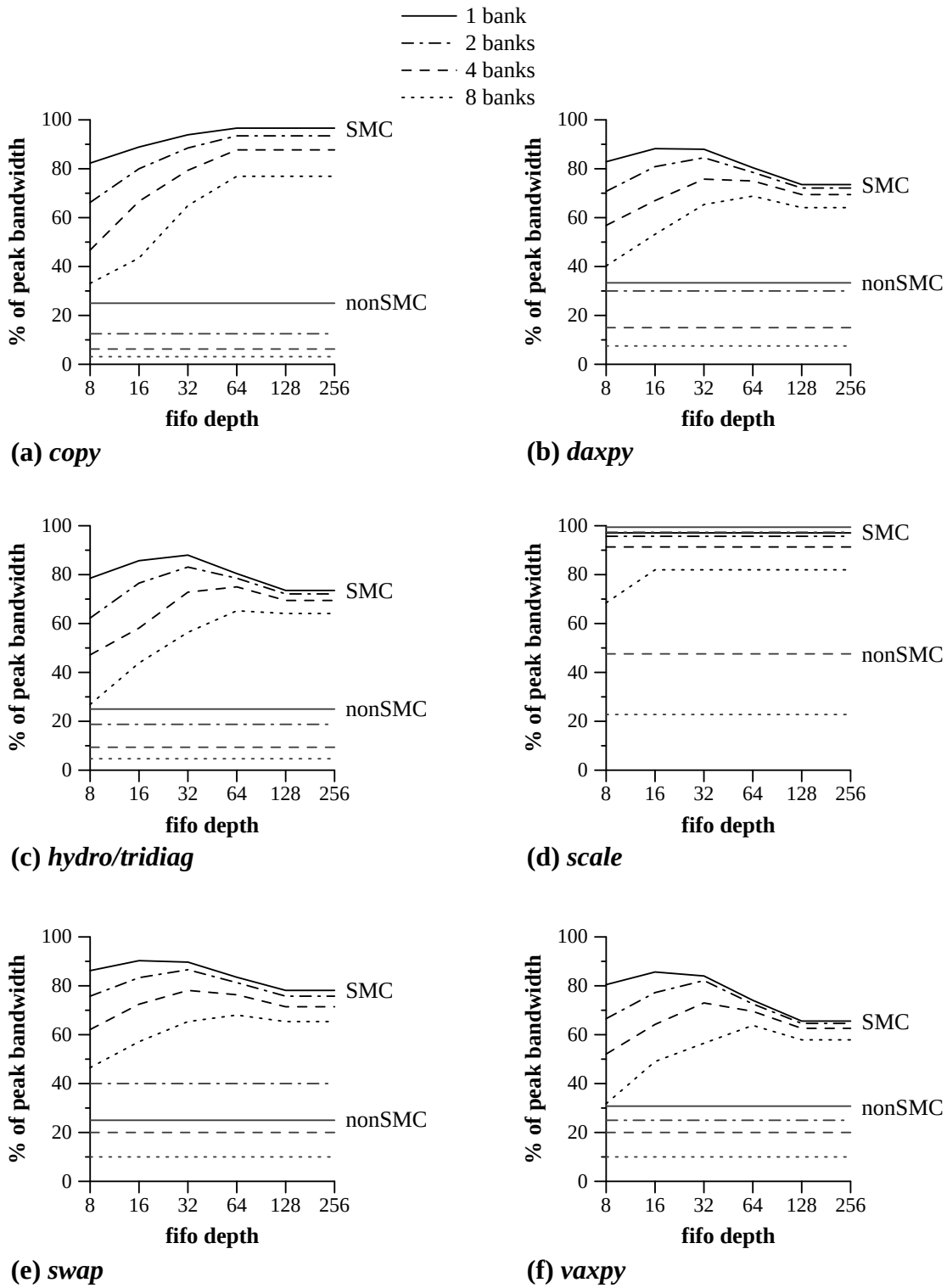


Figure 41 T4 — Medium Vector Performance (for better nonSMC alignment)

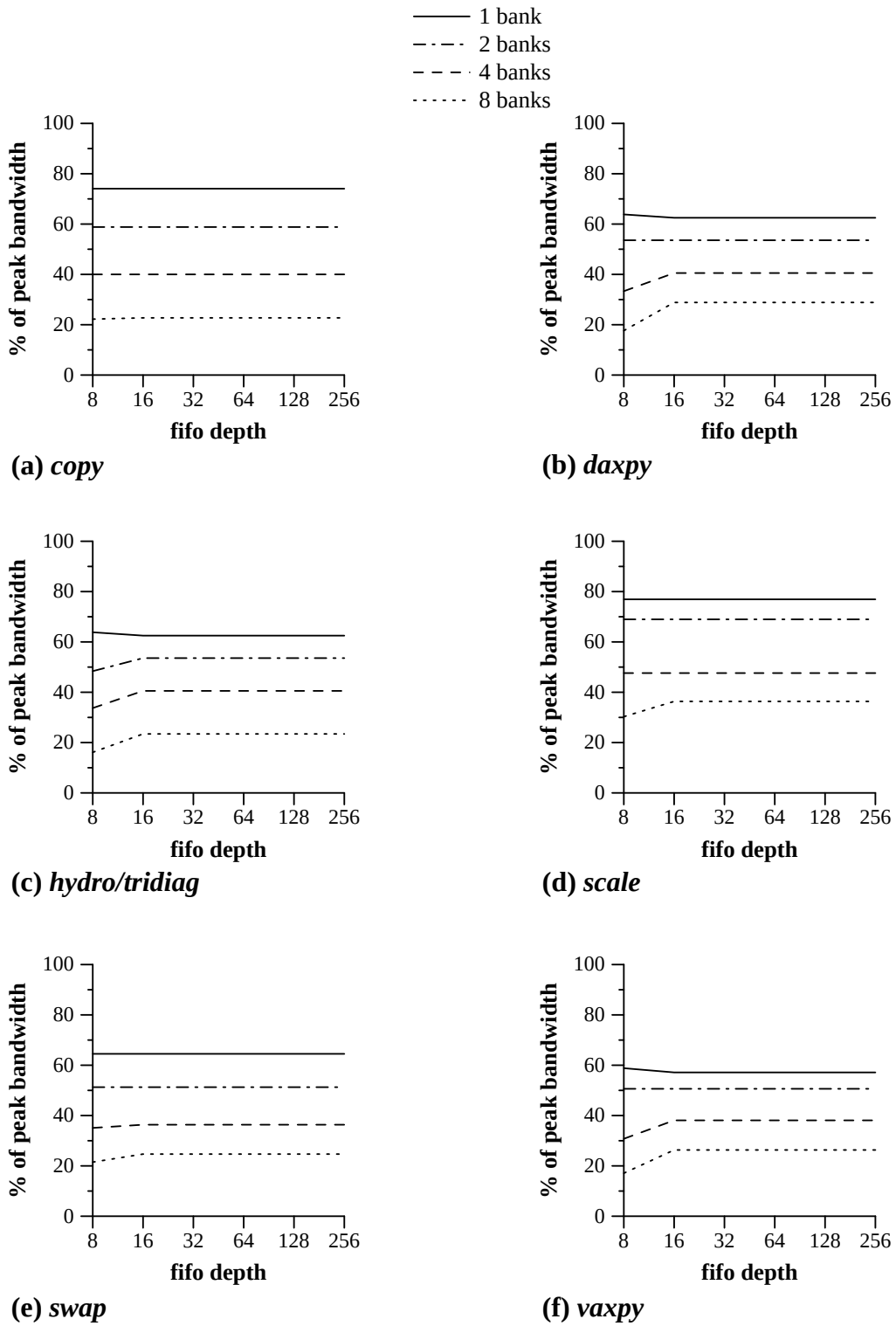


Figure 42 T4 — Short Vector Performance

Table 6 summarizes T4's performance with respect to T1. Recall that blank entries indicate that differences, if any, are less than 0.01%. Numerical values are given for differences of magnitude greater than 1%; entries of lesser magnitude are represented by the sign of the difference.

Table 6: Performance of Scheme T4 with Respect to T1

Differences in Attained Percentage of Peak Bandwidth												
benchmark	banks	medium vectors					long vectors					
		FIFO depth					FIFO depth					
		8	16	32	64	128	8	16	32	64	128	256
copy	1						-		-			
	2						-	-	+	+	+	+
	4						+	+	+	+	+	+
	8											
daxpy	1						-	-	-	-	-	-
	2						-	-	+	+	+	+
	4						+	+	-	+	+	+
	8											
hydro	1						-	-	-	-	-	+
	2	+4.9	+4.2	+1.4			+7.1	+6.5	+5.3	+3.3	+3.0	+2.2
	4	+4.8	+3.0	+2.1			+7.1	+10.2	+7.2	+6.2	+6.5	+5.3
	8	+2.2	+2.9	+2.4	-3.6		+2.7	+9.2	+10.4	+13.2	+10.7	+8.8
scale	1											
	2						+	+	+	+	+	+
	4						+	+	+	+	+	+
	8											
swap	1						-	-	-	-	-	-
	2	-					+	-	+	+	+	+
	4						+	+	+	+	+	+
	8	-2.3					-3.8			-		-
vaxpy	1						-	-	-	-	+	+
	2						+	+	+	+	+	+
	4						+	+	+	+	+	+
	8											

7.1.5 Group 5 — Algorithms P5, R5, and T5

We saw in the last section that a simpler ordering algorithm may yield better performance. How simple can we make the scheme and still achieve high bandwidth? To test this, we implemented an ordering scheme that doesn't even look for accesses that hit a bank's current DRAM page. These algorithms merely issue accesses for the current FIFO until no more ready accesses remain, then they move on to the next FIFO in round-robin order. Figure 43 through Figure 51 illustrate the performance of this group of algorithms.

Figure 43, Figure 46, and Figure 49 illustrate long vector SMC performance. The curves for *copy* and *scale* are virtually identical to those for the Group 1 algorithms. On the *hydro* benchmark, performance is identical to that of the Group 4 schemes. For *daxpy, swap*, and *vaxpy* using shallower FIFOs, the performance for eight-bank memory systems is worse than that for the Group 1 schemes — up to 17.1% of attainable bandwidth less for T5 on *swap* using eight-deep FIFOs and an eight-bank system, or a 13% of peak drop over T4's performance. For deeper FIFOs, i.e. when FIFO depth is scaled with the interleaving factor, performance differences are small.

Medium vector SMC performance is depicted in Figure 44, Figure 47, and Figure 50. These performance curves exhibit similar trends as those for long vectors when compared with the corresponding curves for Group 1. *Daxpy* and *vaxpy* fare slightly worse for shallow FIFOs, and *swap*'s performance is slightly lower overall. Again, these algorithms achieve a higher percentage of peak bandwidth on the *hydro* benchmark, but performance drops slightly (3.6% of peak) for depth-64 FIFOs and an eight-bank memory. For deep FIFOs, performance for all benchmarks converges to that achieved by the other selection algorithms.

Short vector performance is almost identical to that of the corresponding algorithms in Group 4, except for a slight drops for eight banks and shallow FIFOs.

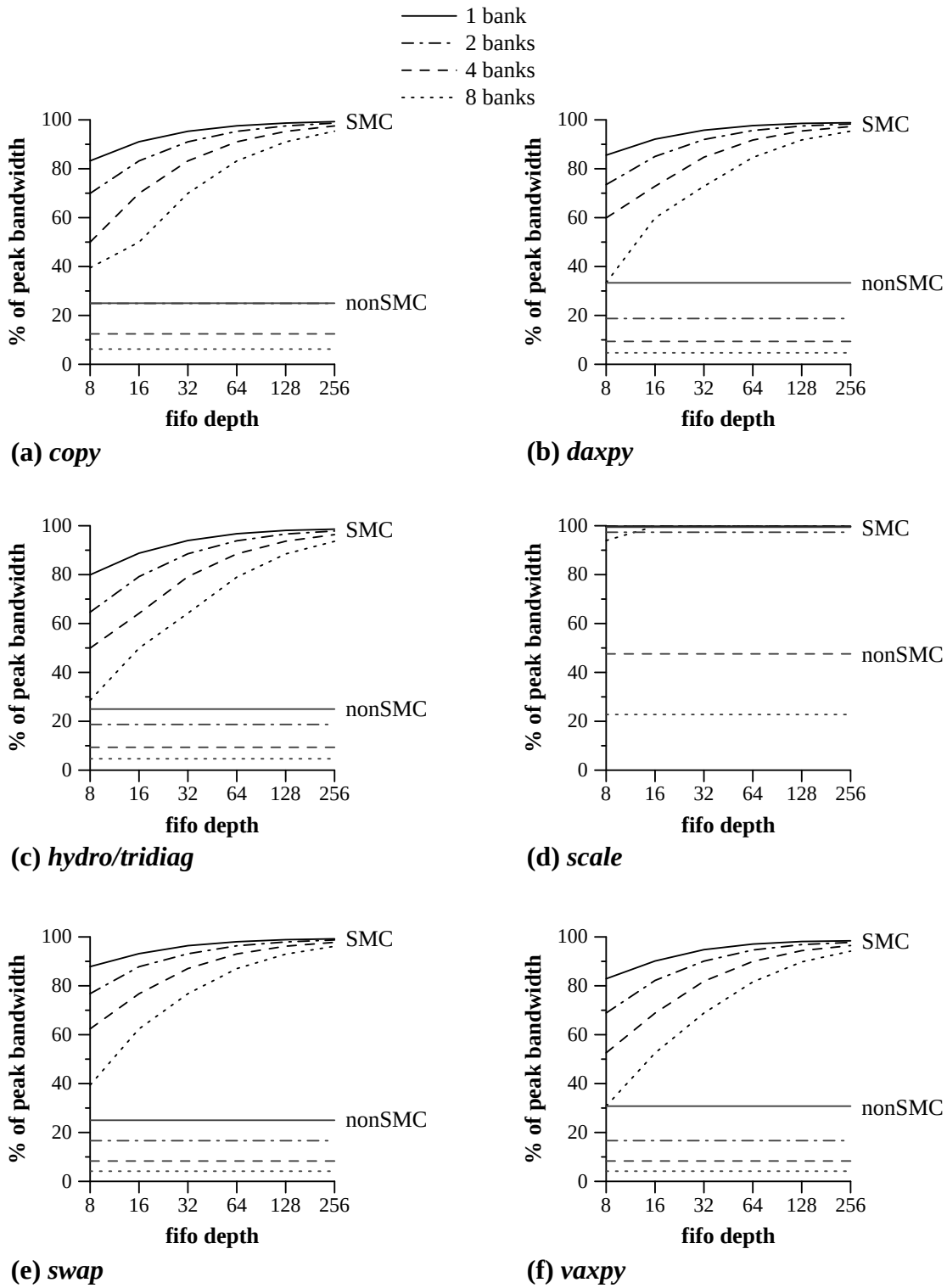


Figure 43 P5 — Long Vector Performance

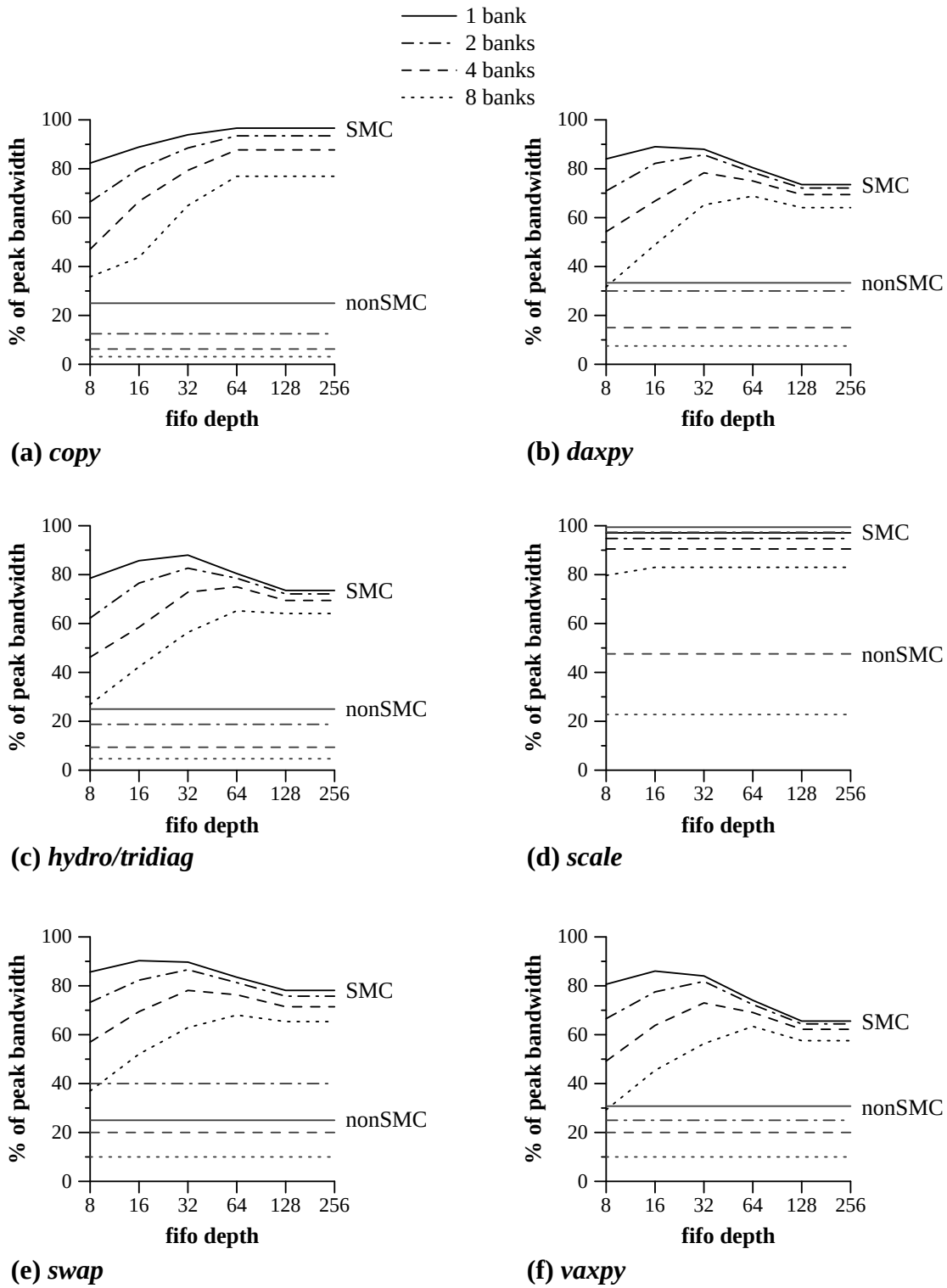


Figure 44 P5 — Medium Vector Performance (for better nonSMC alignment)

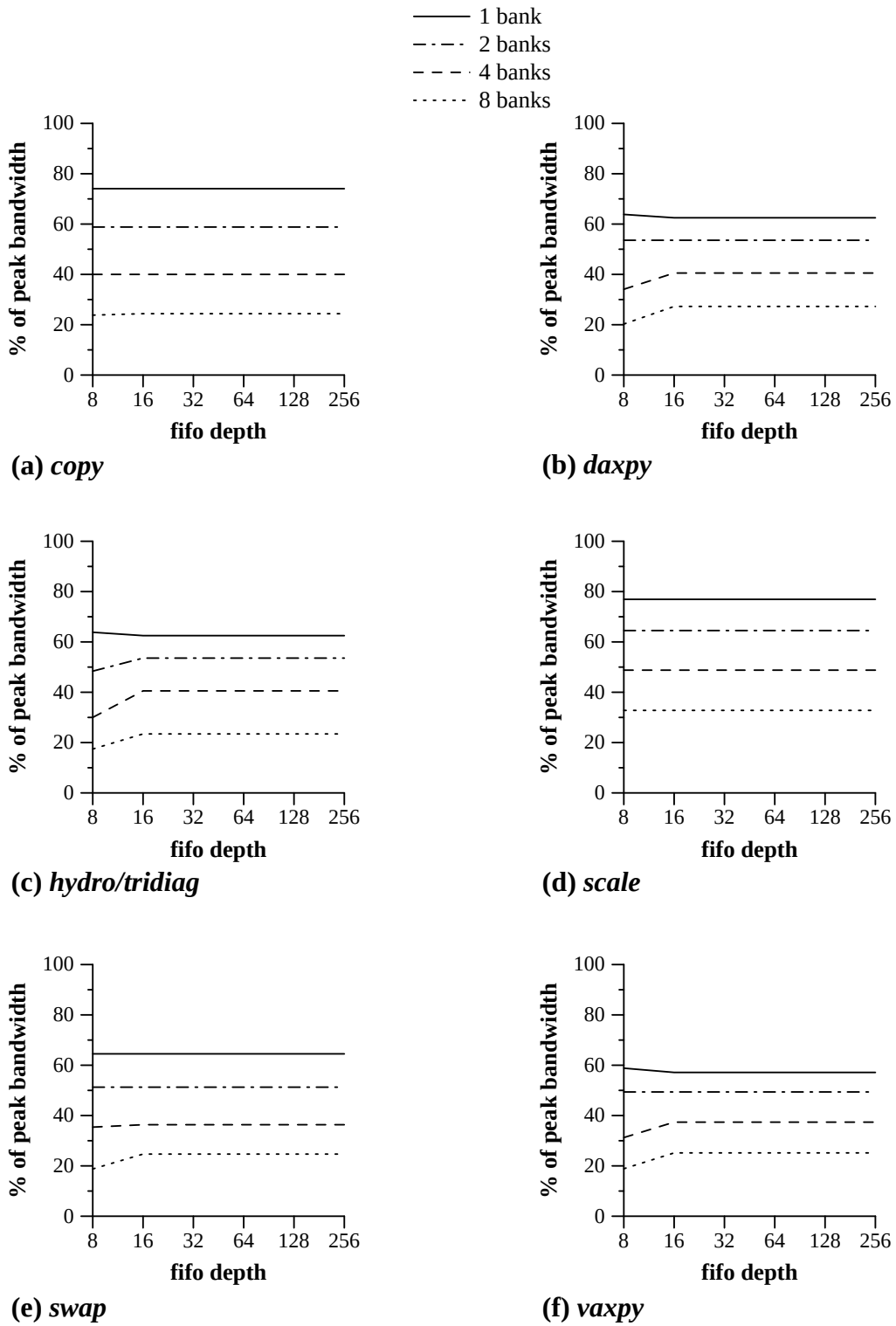


Figure 45 P5 — Short Vector Performance

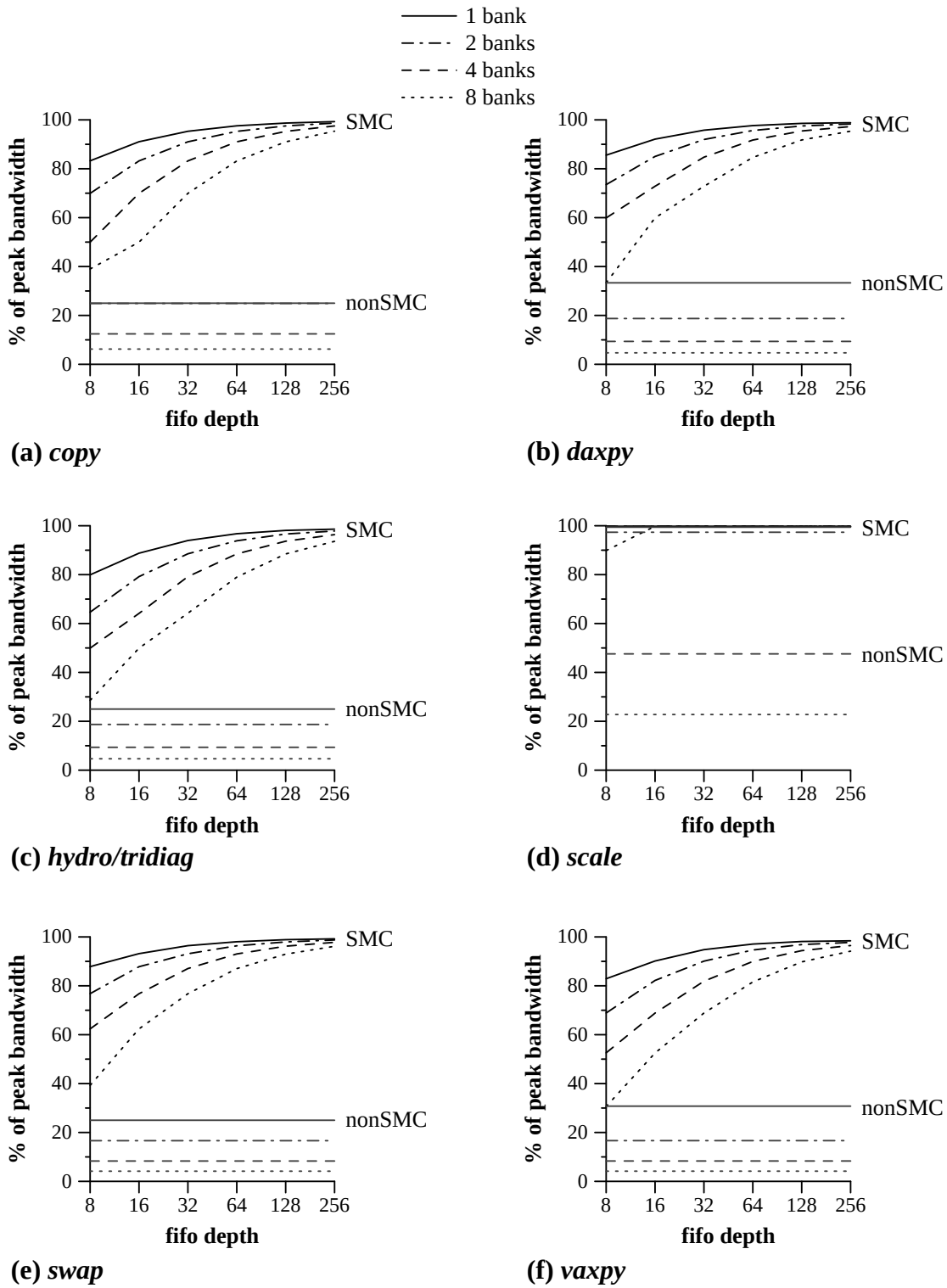


Figure 46 R5 — Long Vector Performance

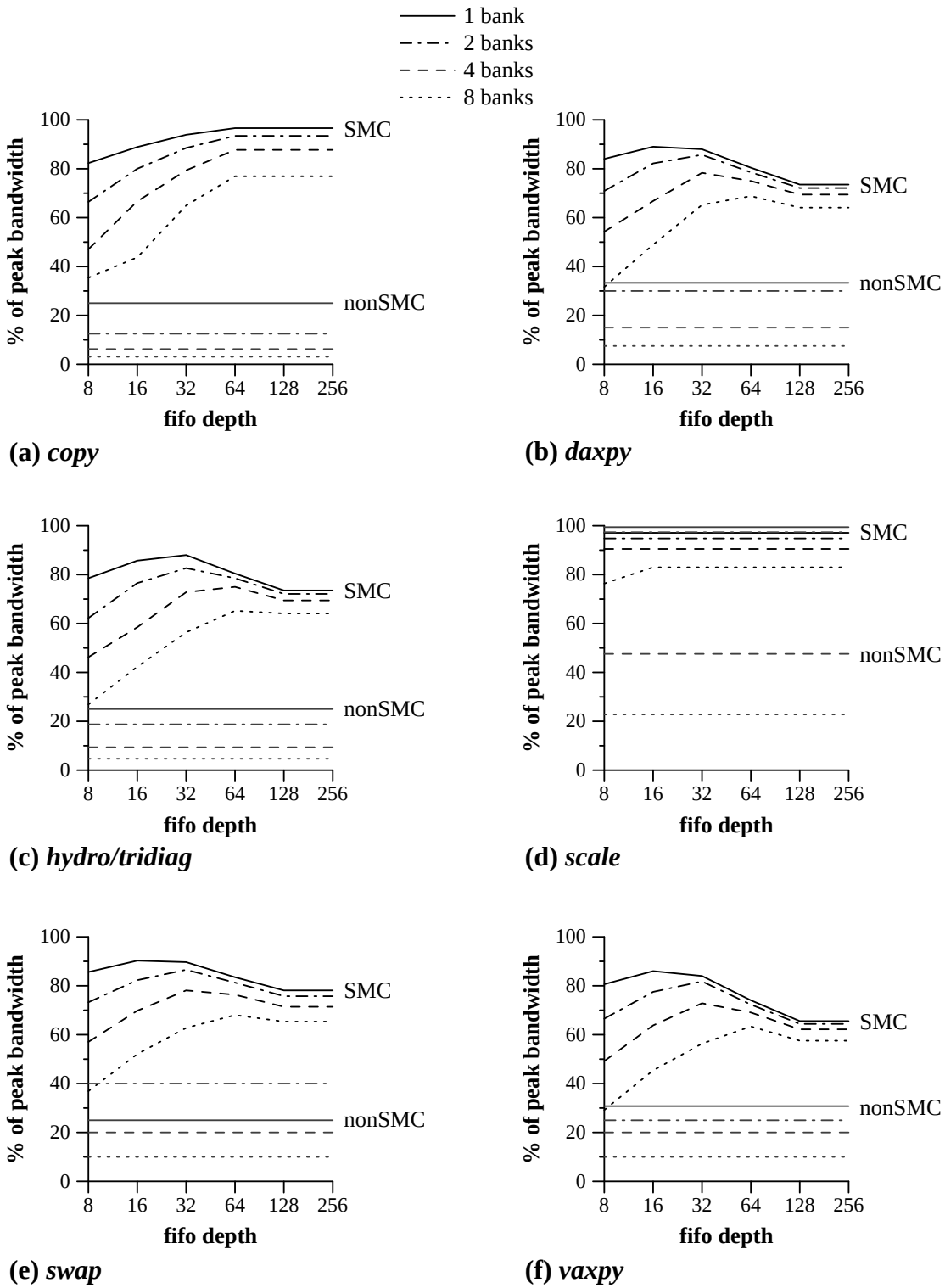


Figure 47 R5 — Medium Vector Performance (for better nonSMC alignment)

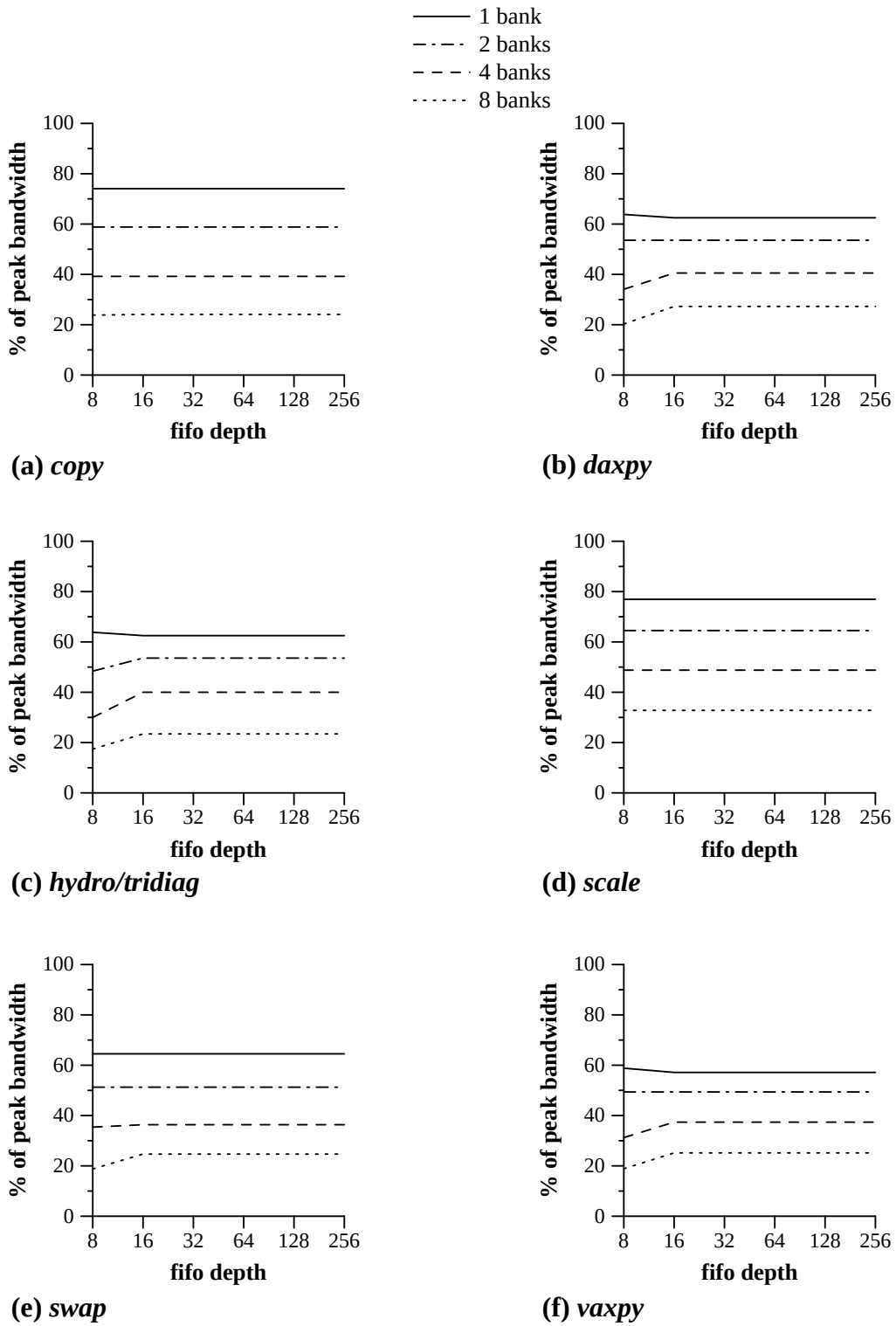


Figure 48 R5 — Short Vector Performance

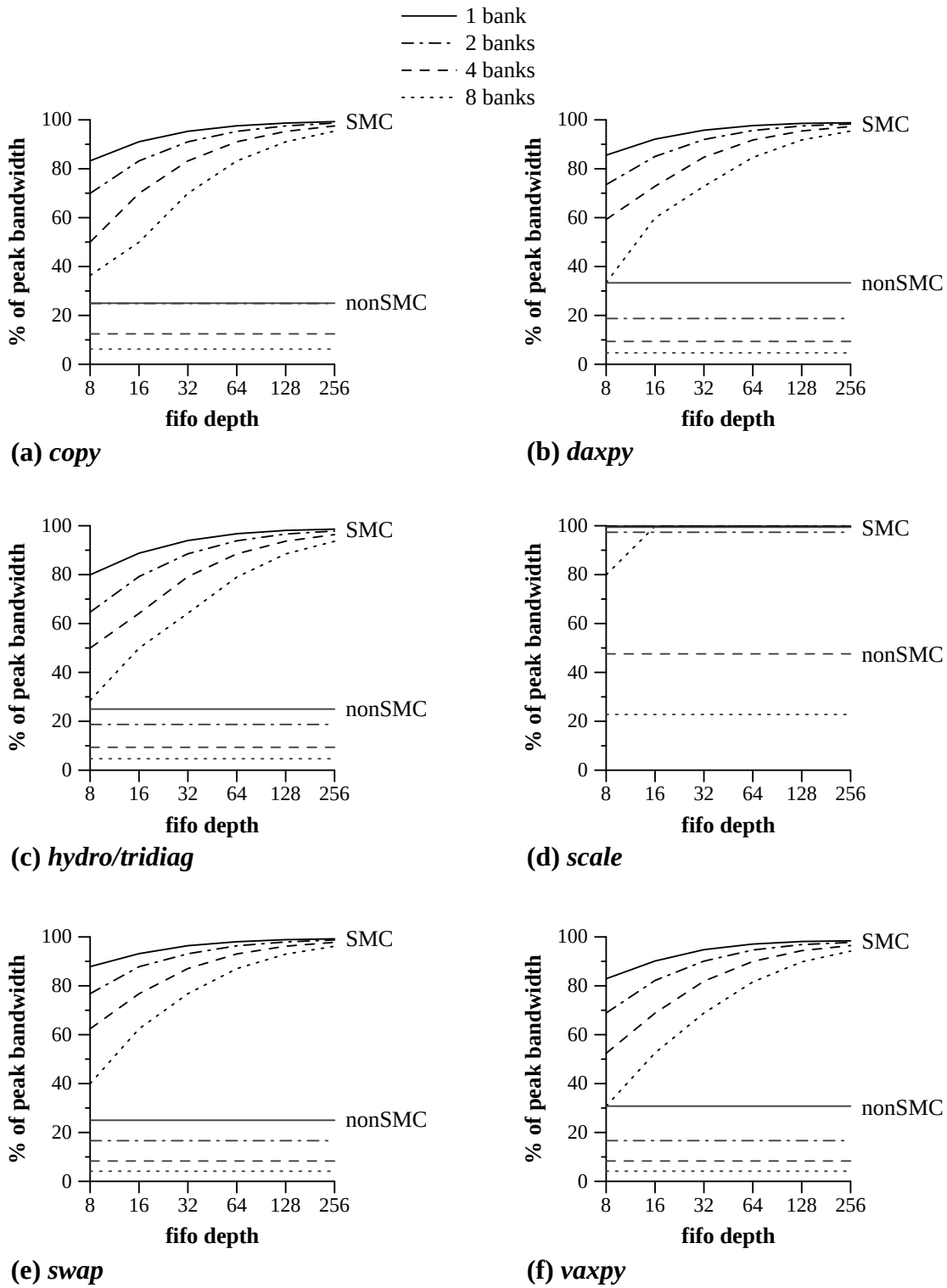


Figure 49 T5 — Long Vector Performance

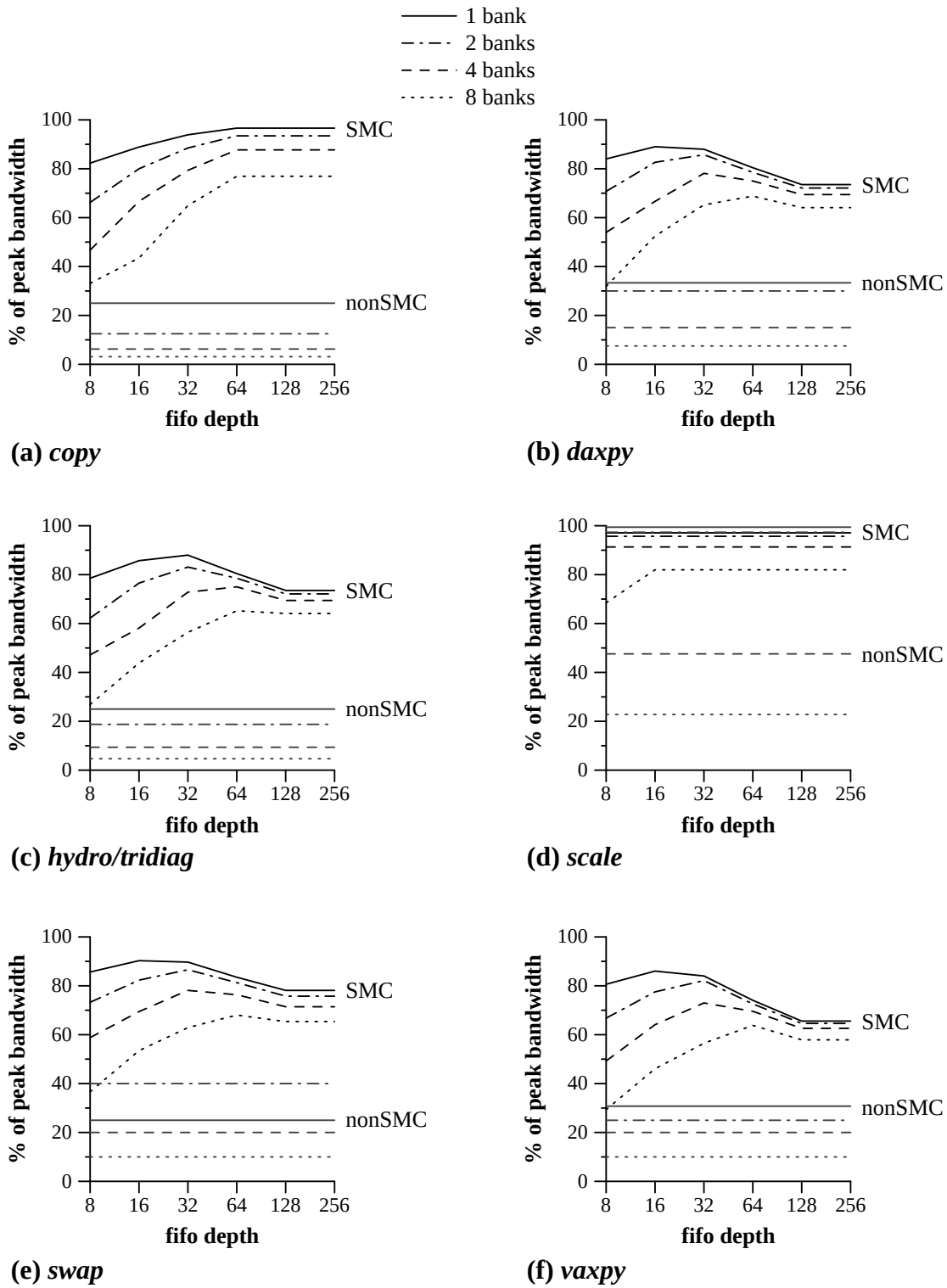


Figure 50 T5 — Medium Vector Performance (for better nonSMC alignment)

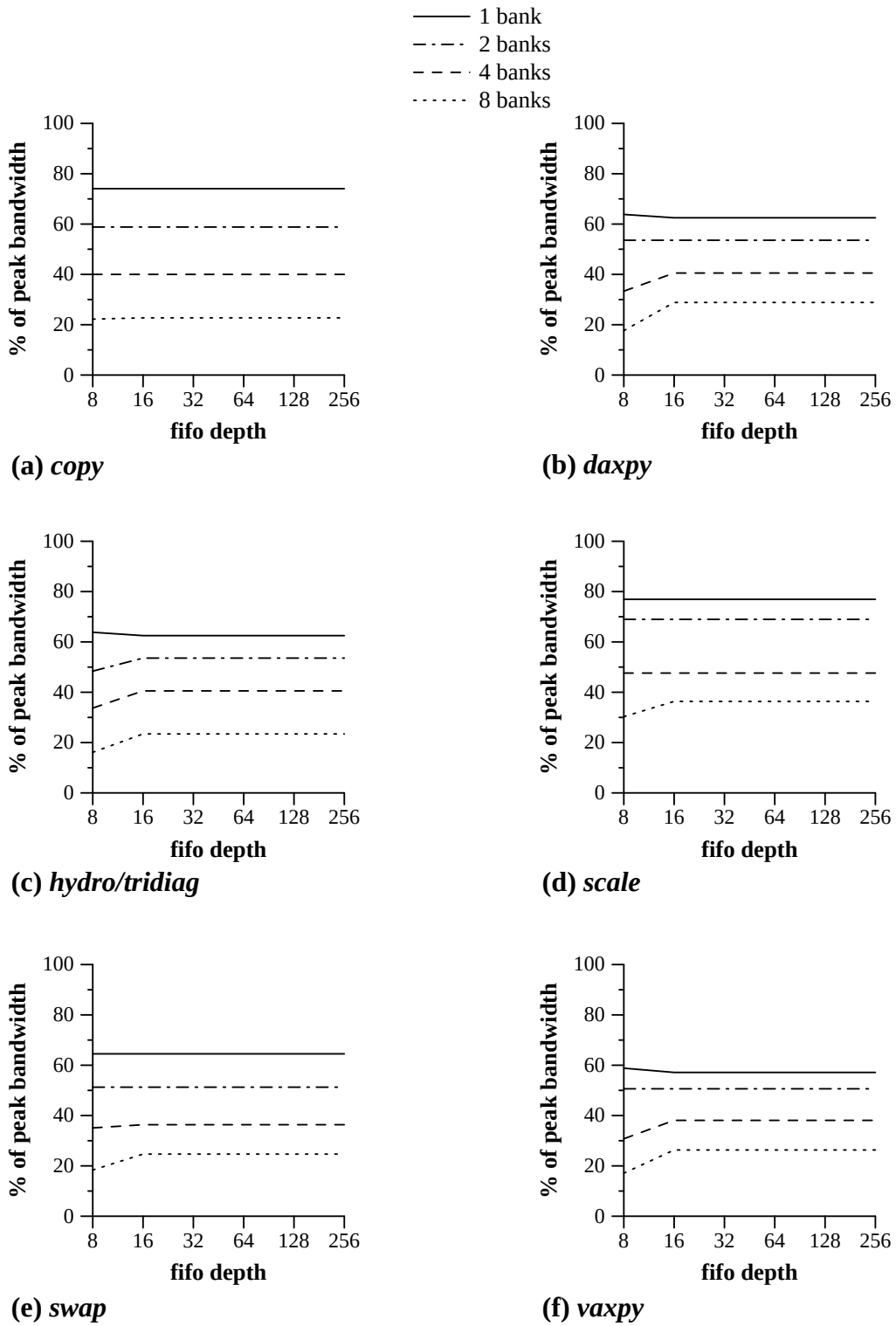


Figure 51 T5 — Short Vector Performance

Table 7 summarizes T5's performance with respect to T1. Recall that blank entries indicate that differences, if any, are less than 0.01%. Numerical values are given for differences of magnitude greater than 1%; entries of lesser magnitude are represented by the sign of the difference.

Table 7: Performance of Scheme T5 with Respect to T1

Differences in Attained Percentage of Peak Bandwidth												
benchmark	banks	medium vectors					long vectors					
		FIFO depth					FIFO depth					
		8	16	32	64	128	8	16	32	64	128	256
copy	1						-		-			
	2						-	-	+	+	+	+
	4						+	+	+	+	+	+
	8											
daxpy	1	+1.2	+				-	-	-	+	+	+
	2		+1.8	+1.2			+	-	+	+	+	+
	4	-2.9	-	+2.4			-	-	-	-	+	+
	8	-8.4	-	-			-12.8	+	-	-		+
hydro	1						-	-	-	-	-	+
	2	+4.9	+4.2	+1.4			+7.1	+6.5	+5.3	+3.3	+3.0	+2.2
	4	+4.8	+3.0	+2.1			+7.1	+10.2	+7.2	+6.2	+6.5	+5.3
	8	+2.2	+2.9	+2.4	-3.6		+2.7	+9.2	+10.4	+13.2	+10.7	+8.8
scale	1											
	2							+		+		
	4							+	+	+	+	+
	8											
swap	1	-					-	-	-	-	-	+
	2	-3.4	-1.0				-3.0	-	-	-	+	+
	4	-3.3	-3.0				-4.2	-3.1	-1.1	-	-	+
	8	-12.1	-3.7	-2.5			-17.1	-4.2	-3.1	-1.1	-	-
vaxpy	1	+	+				-	-	-		+	+
	2	+	+				+	+	+	+	+	+
	4	-2.8	-				-3.8	+	-	+	+	+
	8	-2.4	-2.9				-4.0	-4.5		-	-	+

7.1.6 Group 6 — Algorithms A1 and A2

The algorithms discussed thus far generate memory accesses by first choosing a bank (or banks) to access, and then choosing the appropriate FIFO (or FIFOs) for which to initiate accesses. The algorithms in Group 6 perform their duties in the opposite order: first they choose a FIFO to service, and then they choose the bank to access.

Algorithm A1 goes round-robin through the FIFOs, initiating accesses for the current FIFO until it contains no ready accesses. At that point, the SMC advances to the next FIFO and proceeds to initiate accesses for it. While servicing a particular FIFO, if the next ready access from that FIFO is to a busy bank, the SMC waits until the bank is idle; it does not try to find an access to a currently idle bank. Simulation results for this ordering scheme are depicted in Figure 52 through Figure 54.

Algorithm A2 is a slightly more sophisticated version of A1, incorporating a threshold similar to that of the algorithms in Group 3. If the SMC determines that the next access from the current FIFO will generate a DRAM page miss, it decides whether or not to switch to a different FIFO. When it must issue an access that misses a bank's current page, it attempts to choose the access from a FIFO that contains ready accesses equal to at least half its depth. If the current FIFO requires enough service, the access is issued for it. Otherwise the SMC looks at the next FIFO in sequence, and so on. If no FIFOs meet the threshold, the algorithm issues no accesses at that time. Performance of this algorithm is illustrated in Figure 55 through Figure 57. As expected, simulation results for these algorithms exhibit the same degradation in performance that we saw with many of the other algorithms for shallow FIFOs on memory systems with a high degree of concurrency.

For long vectors, performance tends to be lower than that of Algorithm P1 for most benchmarks run with FIFOs up to 32 double-words deep. *Hydro* is the exception to this: Algorithm A1 outperforms the Group 1 schemes for all FIFO depths and memory systems.

For deeper FIFOs, A1's performance for all benchmarks is within a few percent of that for the Group 4 algorithms, but for shallow FIFOs (especially on a memory system with many banks), its performance dips to 16.9% of peak less.

For medium vectors, performance again tends to be lower than that of the Group 1 algorithms for FIFOs of depth eight, sixteen, and thirty-two. When compared with Group 4, these algorithms provide virtually identical performance for deeper FIFOs, but performance is often over 10% of peak worse for shallow FIFOs and higher interleavings.

Short vector performance is similar to that of Algorithm P1, but A1 performed worse in a few instances. Most benchmarks fare worse with eight-deep FIFOs, regardless of the number of banks in the memory system. A1's performance on the *swap* kernel on a two-bank system is about 5% of peak below that of P1's.

On long vectors, A2 performs almost identically to A1. On medium vectors, however, A2 fares significantly worse for deeper FIFOs on the *copy* benchmark. Smaller drops in performance are evident for the *swap* and *hydro* benchmarks for FIFOs of sixty-four or more double-words. Medium vector performance for the other benchmarks is about the same as for A1, with performance generally dropping by less than 2% of attainable bandwidth. On short vectors, the bandwidth delivered by A2 on the *copy* benchmark is much lower — almost 20% of peak difference for a single-module system. A2 performs about the same as A1 on the *scale* benchmark, and performance for the two algorithms is similar for the *daxpy, vaxpy,* and *swap* kernels with FIFOs at least sixteen deep. A2 consistently outperforms A1 for very shallow FIFOs and 8-bank interleavings, and for *swap* in general on all but the single-bank memory (but only by one or two percent of peak, in the latter case). Neither of these is a strong argument in favor of A2.

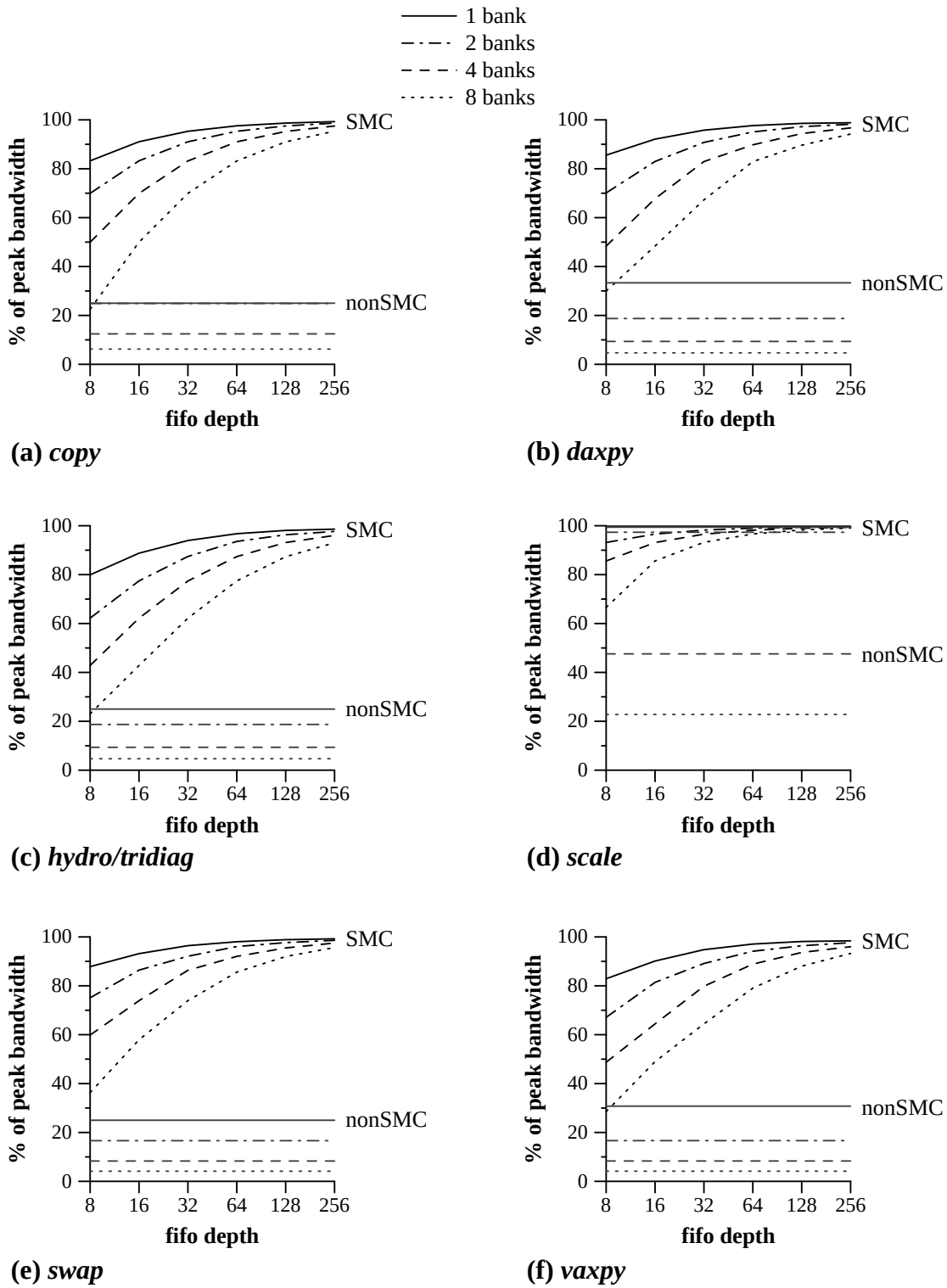


Figure 52 A1 — Long Vector Performance

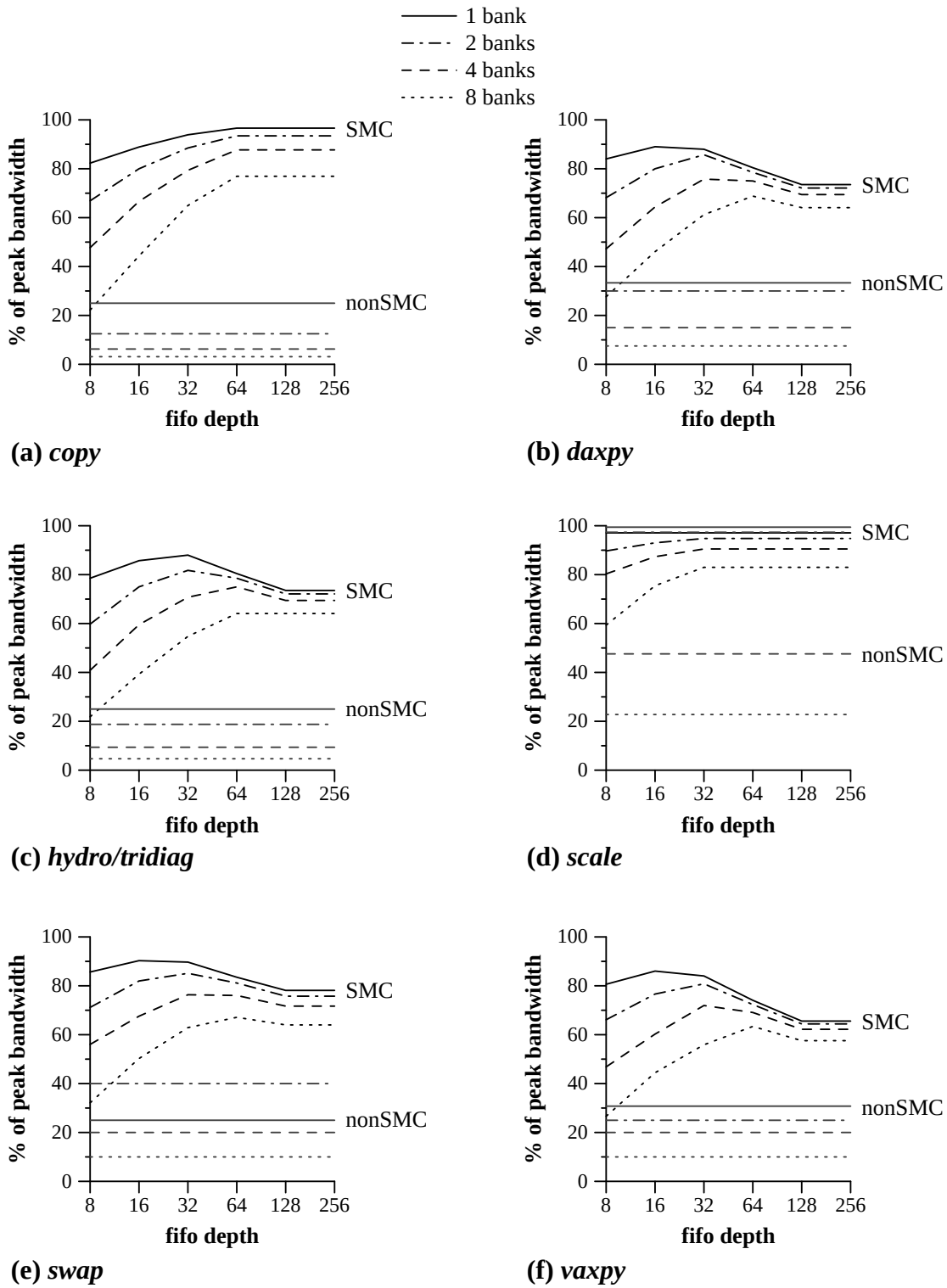


Figure 53 A1 — Medium Vector Performance

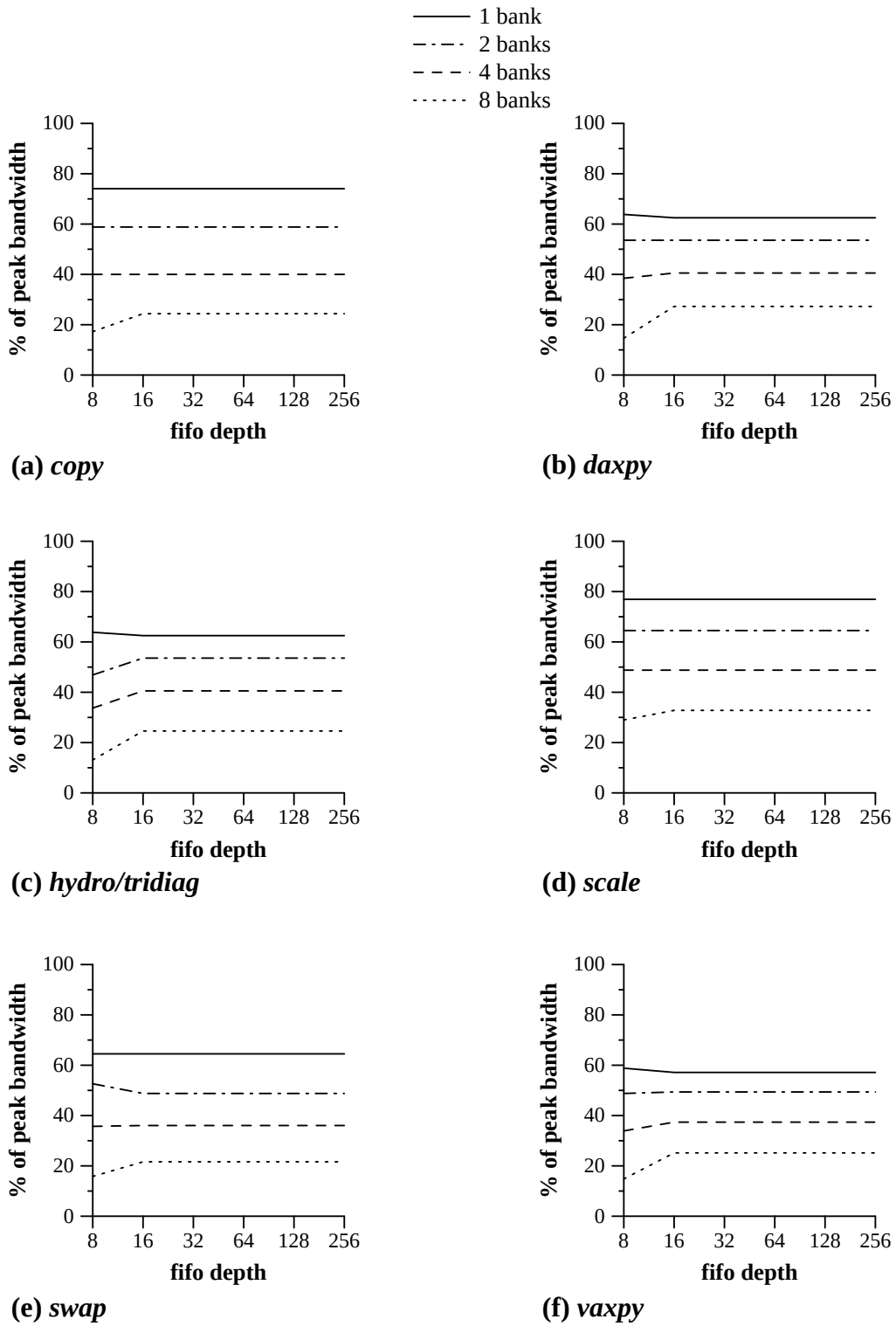


Figure 54 A1 — Short Vector Performance

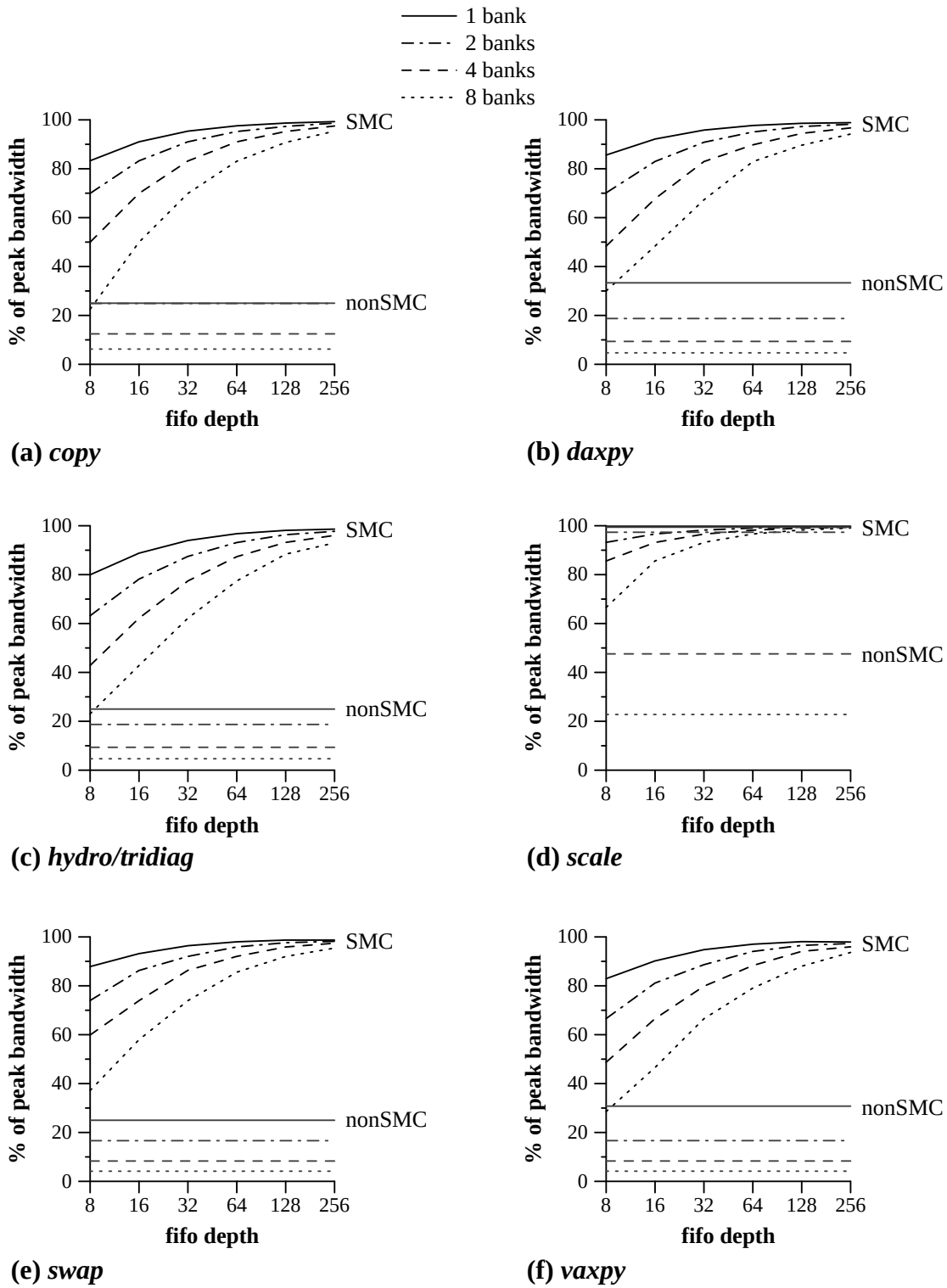


Figure 55 A2 — Long Vector Performance

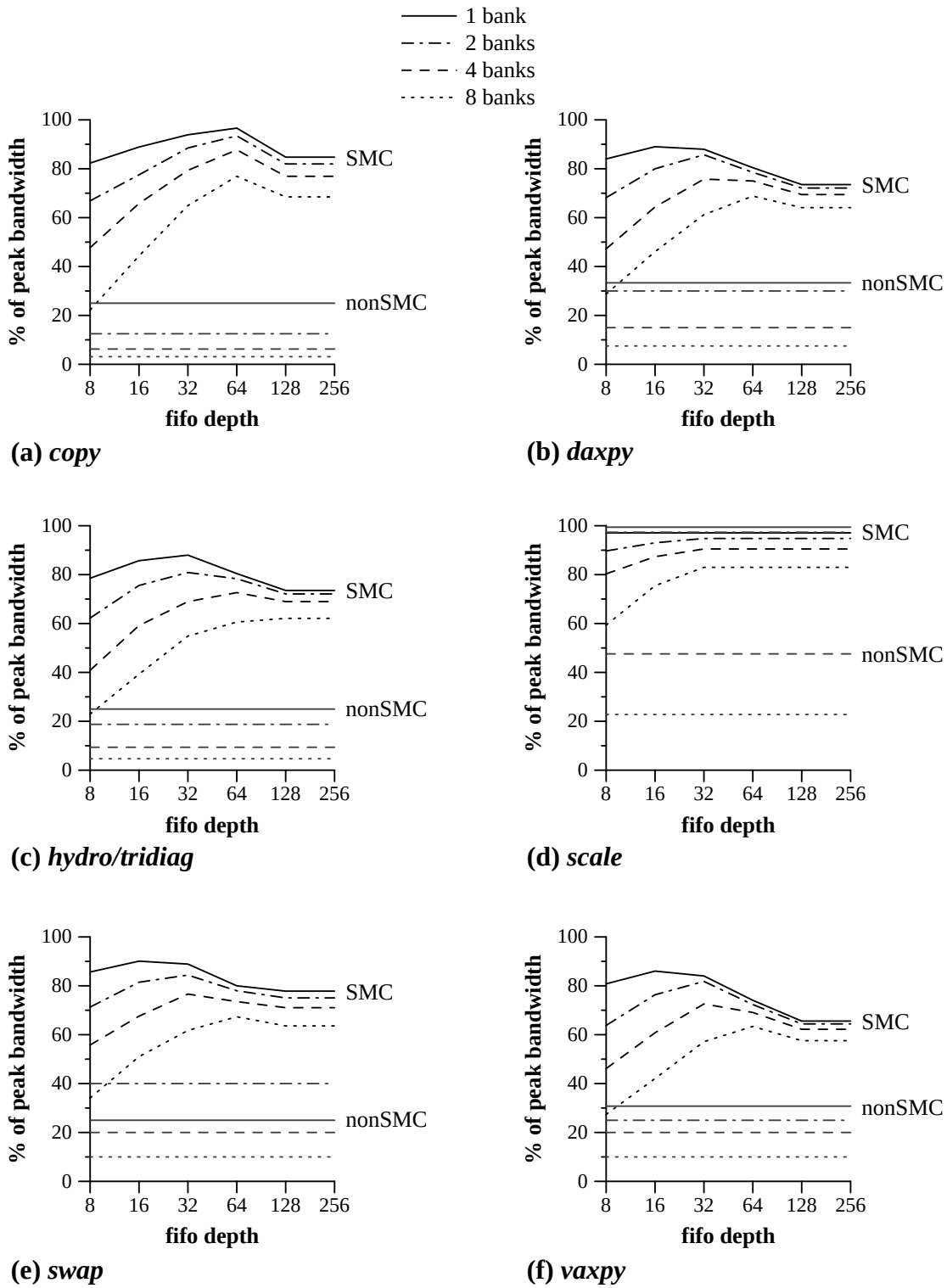


Figure 56 A2 — Medium Vector Performance

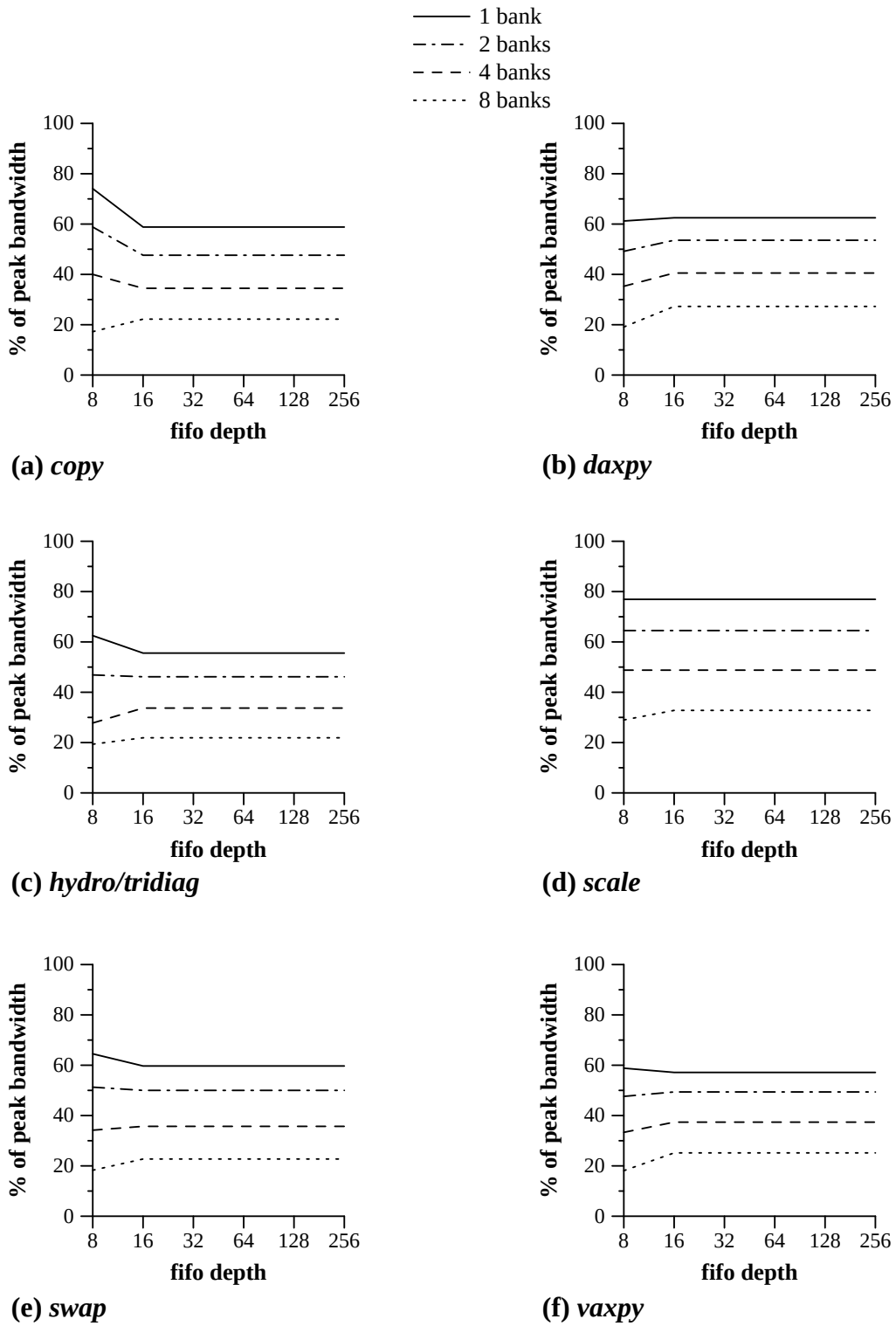


Figure 57 A2 — Short Vector Performance

Table 8 summarizes A1's performance with respect to T1. Table 9 does the same for A2. Recall that blank entries indicate that differences, if any, are less than 0.01%. Numerical values are given for differences of magnitude greater than 1%; entries of lesser magnitude are represented by the sign of the difference.

Table 8: Performance of Scheme A1 with Respect to T1

Differences in Attained Percentage of Peak Bandwidth												
benchmark	banks	medium vectors					long vectors					
		FIFO depth					FIFO depth					
		8	16	32	64	128	8	16	32	64	128	256
copy	1						-		-			
	2	+						+	+	+	+	+
	4	+1.0						+	+	+	+	+
	8	-10.9	+				-13.8	+				
daxpy	1	+1.2	+				-	-	-	+	+	+
	2	-2.6	-	+1.2			-3.4	-2.2	-1.1	-	-	+
	4	-9.6	-2.6				-11.5	-5.4	-2.1	-2.0	-1.0	-
	8	-12.6	-7.1	-4.1			-16.1	-11.5	-6.0	-2.1	-2.1	-
hydro	1						-	-	-	-	-	+
	2	+2.4	+2.7				+4.6	+4.8	+4.2	+3.1	+2.6	+2.0
	4	-1.5	+4.4					+8.3	+5.5	+5.0	+6.0	+5.0
	8	-2.8	-1.7	+	-4.7		-2.8	+2.1	+8.3	+11.7	+9.6	+8.2
scale	1											
	2	-6.0	-2.7	-	-	-	-6.6	-3.2	-1.6	-	-	-
	4	-11.0	-4.0	-	-	-	-14.1	-6.6	-3.2	-1.5	-	-
	8	-9.1	-6.5	+1.0	+1.0	+1.0	-13.3	-14.2	-6.6	-3.2	-1.5	-
swap	1	-					-	-	-	-	-	+
	2	-5.5	-1.4	-1.5	-		-4.7	-2.1	-1.6	-	-	+
	4	-6.1	-4.9	-1.8	-	+	-6.7	-5.9	-1.8	-1.4	-	
	8	-16.6	-6.9	-2.5	-	-1.4	-20.7	-8.6	-5.9	-2.5	-1.6	-
vaxpy	1	+	+				-	-	-		+	+
	2	-	-	-1.3	-	-	-1.7	-	-	-	-	+
	4	-5.2	-4.0	-1.1	-	-	-7.5	-4.4	-2.2	-1.1	-	-
	8	-5.1	-4.6	-	-	-	-6.2	-10.4	-2.3	-2.7	-1.9	-

Table 9: Performance of Scheme A2 with Respect to T1

Differences in Attained Percentage of Peak Bandwidth												
benchmark	banks	medium vectors					long vectors					
		FIFO depth					FIFO depth					
		8	16	32	64	128	8	16	32	64	128	256
copy	1					-11.9		-				
	2	+	-2.5			-11.5		+	+	-	-	+
	4	+1.0	-			-10.8	+	+	+	+	+	+
	8	-10.9	+			-8.4	-13.8		-	-	-	
daxpy	1	+1.2	+				-	-	-	+	+	+
	2	-2.6	-	+1.2			-3.4	-2.2	-1.1	-	-	+
	4	-9.6	-2.6				-11.5	-5.4	-2.1	-2.0	-	-
	8	-11.5	-7.1	-4.1			-16.1	-11.5	-6.0	-2.1	-2.1	-
hydro	1											=
	2	+4.9	+3.3	-	-		+4.6	+4.8	+4.2	+3.1	+2.6	+2.0
	4	-1.5	+4.0	-1.8	-2.4	-		+8.3	+5.5	+5.0	+6.0	+5.0
	8	-1.7	-1.7	+	-8.2	-2.0	-2.8	+2.1	+8.3	+11.7	+10.6	+8.2
scale	1											
	2	-6.0	-2.7	-	-	-	-6.6	-3.1	-1.6	-	-	-
	4	-11.0	-4.0	-	-	-	-14.1	-6.5	-3.1	-1.5	-	-
	8	-9.1	-6.5	+1.0	+1.0	+1.0	-13.3	-14.2	-6.6	-3.2	-1.5	-
swap	1	-	-	-	-3.5	-	-	-	-	-	-	+
	2	-5.3	-1.9	-2.2	-3.3	-	-5.8	-2.2	-1.6	-	-	+
	4	-6.3	-4.9	-1.5	-2.8	-	-6.7	-5.9	-1.9	-1.4	-	
	8	-14.7	-6.1	-3.6	-	-1.8	-19.9	-8.6	-5.9	-2.5	-1.6	-1.0
vaxpy	1	+	+				+		-	-	-	-
	2	-2.8	-	-	-	-	-2.2	-1.0	-1.4	-	-	+
	4	-5.9	-3.4	-	-	-	-7.5	-2.2	-2.2	-1.7	-	-
	8	-4.4	-7.0	+	-	-	-6.2	-10.4	-2.3	-2.7	-1.9	-

7.2 Performance Summary of the Remaining FIFO-Selection Schemes

As we saw in the last section, performance of the different access ordering schemes tends to be very similar. Here we summarize the performance of the remaining five FIFO-selection algorithms (6 through 10) when paired with the T bank-selection scheme. Table 4 through indicate relative performance of these schemes as compared to Scheme T1.

Recall that Algorithm 1 continues to issue accesses to the current DRAM page for the current bank, *b*, until no more such ready accesses exist. It then looks for the FIFO requiring the most service from *b*, considering first the FIFO last accessed by *b*. Algorithm 6 is identical to Algorithm 1, except that the search for the FIFO requiring the most service from the current bank begins with the last FIFO accessed by *any* bank. Performance of Algorithm 6 is summarized in Table 4. Entries are computed by subtracting T1's performance from T6's. Values are rounded to the nearest percent; values that round to zero are represented simply by either "+" or "-", according to the value's sign; and values with magnitude smaller than one hundredth of one percent are left blank.

Algorithm 7 is similar, except that when a page-miss is inevitable, it chooses the next access from the FIFO requiring the most service from *all* banks, starting the search with the last FIFO accessed by the current bank. Algorithm 8 is identical, except that the search for the FIFO requiring the most service begins with the last FIFO accessed by any bank. Performance for Algorithm 7 is summarized in Table 5. Performance of Algorithm 8 is summarized in Table 6.

Algorithm 10 resembles Algorithm 5: neither explicitly tries to initiate accesses that hit the current DRAM page. Algorithm 10 issues the next access it finds, and considers the FIFOs in round-robin order beginning with the last FIFO accessed by any bank. Algorithm 5 begins its search with the last FIFO accessed by the current bank. Algorithm 10's performance is summarized in Table 8.

Table 10: Performance of Scheme T6 with Respect to T1

Differences in Attained Percentage of Peak Bandwidth												
benchmark	banks	medium vectors					long vectors					
		FIFO depth					FIFO depth					
		8	16	32	64	128	8	16	32	64	128	256
copy	1											
	2											
	4											
	8											
daxpy	1											
	2							+				
	4						-					
	8											
hydro	1											
	2											
	4											
	8							+				
scale	1											
	2											
	4											
	8	-8.3					-13.3					
swap	1											
	2		+1.1				-	+	+	-	-	
	4						-	+	+	+		
	8									+	+	
vaxpy	1											
	2						+	+	-			
	4						+	+	+	+	+	-
	8	-1.2					-			+		

Table 11: Performance of Scheme T7 with Respect to T1

Differences in Attained Percentage of Peak Bandwidth												
benchmark	banks	medium vectors					long vectors					
		FIFO depth					FIFO depth					
		8	16	32	64	128	8	16	32	64	128	256
copy	1											
	2						-	+	+	+	+	+
	4						+	+	+	+	+	+
	8											
daxpy	1											
	2	+1.4	+	-			+1.4	+	+	+	+	+
	4	-2.5	+	+1.8			-1.7	+	-	+	+	+
	8	-4.4	-3.3	-3.4			-6.8	-4.4	-	-		+
hydro	1											
	2	+3.4	+4.2	+2.3			+7.0	+6.5	+5.2	+3.3	+2.8	+1.7
	4	-1.4	+1.2	-1.9			-1.3	+4.6	+7.1	+5.9	+5.8	+5.0
	8	+1.5	-1.9		-3.6		+2.6	+	+1.0	+13.2	+7.5	+4.8
scale	1											
	2						+	+	+	+	+	+
	4						+	+	+	+	+	+
	8											
swap	1											
	2	-					+	-	+	+	+	+
	4		-1.5					-	-	+	+	+
	8	-6.6					-11.2			-	-	-
vaxpy	1											
	2	-2.0		-4.8	-	-	-1.1	+	+	-	-	+
	4	-6.8	-7.7	-4.9	-		-6.6	-6.5	-3.9	-	-	-
	8	-1.7	-5.9	-2.3	+	+	-2.9	-10.2	-5.8	-2.2	-	-2.4

Table 12: Performance of Scheme T8 with Respect to T1

Differences in Attained Percentage of Peak Bandwidth												
benchmark	banks	medium vectors					long vectors					
		FIFO depth					FIFO depth					
		8	16	32	64	128	8	16	32	64	128	256
copy	1											
	2						-	+	+	+	+	+
	4						+	+	+	+	+	+
	8											
daxpy	1											
	2	+1.4	+	-			+1.4	+	+	+	+	+
	4	-1.6		+1.6			+	+1.0	-	+	-	+
	8	-9.1	-2.4	-3.3			-12.8	+	-	-		+
hydro	1											
	2	+4.1	+4.2	+2.3			+7.3	+6.5	+5.2	+3.3	+2.8	+1.7
	4	-	+2.7	-3.8			-	+3.6	+6.9	+5.9	+5.8	+5.2
	8	+1.1	-1.7	-4.3	-3.6		+2.6	+	+1.0	+13.2	+10.3	+5.5
scale	1											
	2						+	+	+	+	+	+
	4						+	+	+	+	+	+
	8	-8.3					-13.3					
swap	1											
	2						+	-	+	+	+	+
	4							+	+1.1	+	+	+
	8	-6.6					-10.0			+	+	+
vaxpy	1											
	2	-1.8	+	-4.8	-	-	-1.1	+	+	-	-	+
	4	-7.4	-4.7	-5.3			-6.6	-6.5	-3.9	-	-	-
	8	-1.8	-6.3	-4.1	-2.3		-2.9	-10.2	-5.8	-2.2	-	-2.4

Algorithm 9 resembles Algorithm 4, in that it tries to issue accesses that hit the current DRAM page, but when it cannot, it chooses the next access found. Algorithm 4 begins its

search for this access with the last FIFO accessed by the current bank; Algorithm 9 begins with the last FIFO accessed by any bank. Table 7 summarizes this algorithm's performance.

Table 13: Performance of Scheme T9 with Respect to T1

Differences in Attained Percentage of Peak Bandwidth												
benchmark	banks	medium vectors					long vectors					
		FIFO depth					FIFO depth					
		8	16	32	64	128	8	16	32	64	128	256
copy	1						-		-			
	2							+	+	+	+	+
	4						+	+	+	+	+	+
	8											
daxpy	1						-	-	-	-	-	-
	2						+	-	+	+	+	+
	4							+		+	+	+
	8	+		-1.3			-					
hydro	1						-	-	-	-	-	+
	2	+4.9	+4.2	+1.4			+7.2	+6.5	+5.3	+3.3	+3.0	+2.2
	4	+4.8	+3.0	+2.1			+7.1	+10.2	+7.2	+6.2	+6.5	+5.3
	8	+	+2.9	+2.4	-3.6		+	+9.2	+10.4	+13.2	+10.8	+8.8
scale	1											
	2						+	+	+	+	+	+
	4						+	+	+	+	+	+
	8	-8.3					-13.3					
swap	1						-	-	-	-	-	-
	2	-					+	+	+	+	+	+
	4			+1.9				+	+	+	+	+
	8	-3.9	-				-6.8			+	+	
vaxpy	1						-	-	-	-	+	+
	2	-1.2					+	+	+	+	+	+
	4						+	+	-1.4	-	+	+
	8	-1.4		-2.4			+		-	+		

Table 14: Performance of Scheme T10 with Respect to T1

Differences in Attained Percentage of Peak Bandwidth												
benchmark	banks	medium vectors					long vectors					
		FIFO depth					FIFO depth					
		8	16	32	64	128	8	16	32	64	128	256
copy	1						-		-			
	2						-2.8	-1.2	+	-	+	+
	4						+	+	+	+	+	+
	8											
daxpy	1	+1.2	+				-	-	-	-	+	+
	2	-1.6	+	+1.2			-4.5	-	+	+	+	+
	4	-4.8	-3.7	-1.1			-5.5	-2.9	-	-	+	+
	8	-11.9	-4.2	-7.2	-4.7		-17.6	-4.4	-2.0	-	-	0.2
hydro	1						-	-	-	-	-	+
	2	+4.0	+4.2	+1.4			+5.8	+6.0	+5.1	+3.3	+2.90	+2.2
	4	-7.7	+3.7	+2.1			-2.3	+8.5	+6.5	+5.9	+6.5	+5.3
	8	-2.3	-7.7	-2.9	-10.7		-2.8	-	+8.5	+12.7	+10.4	+8.7
scale	1											
	2						+	+	+	+	+	+
	4						+	+	+	+	+	+
	8	-8.3					-13.3					
swap	1	-					-	-	-	-	-	+
	2	-3.4	-1.0				-3.0	-1.7	-	-	+	+
	4	-10.8	-7.1				-8.7	-4.0	-1.5	-	-	+
	8	-19.9	-10.6	-8.2	-8.0	-10.1	-29.2	-10.2	-15.6	-1.4	-	-
vaxpy	1	+	+				-	-	-		+	+
	2	+	+	+	+		-5.4	-6.2	+	+	+	+
	4	-9.0	-10.4		+		-11.7	-5.0	-	-	+	+
	8	-4.3	-14.8	-3.0	-3.9	+	-6.2	-10.2	-5.0	-	-	+

Algorithms 1, 2, 3, 4, 5, and 7 thus use a local FIFO priority, whereas the other algorithms use a global FIFO priority. Likewise, schemes 1, 2, 3, and 6 uses local (subFIFO) status

information to choose the next “best” access. The others use global (FIFO) status information to make this decision.

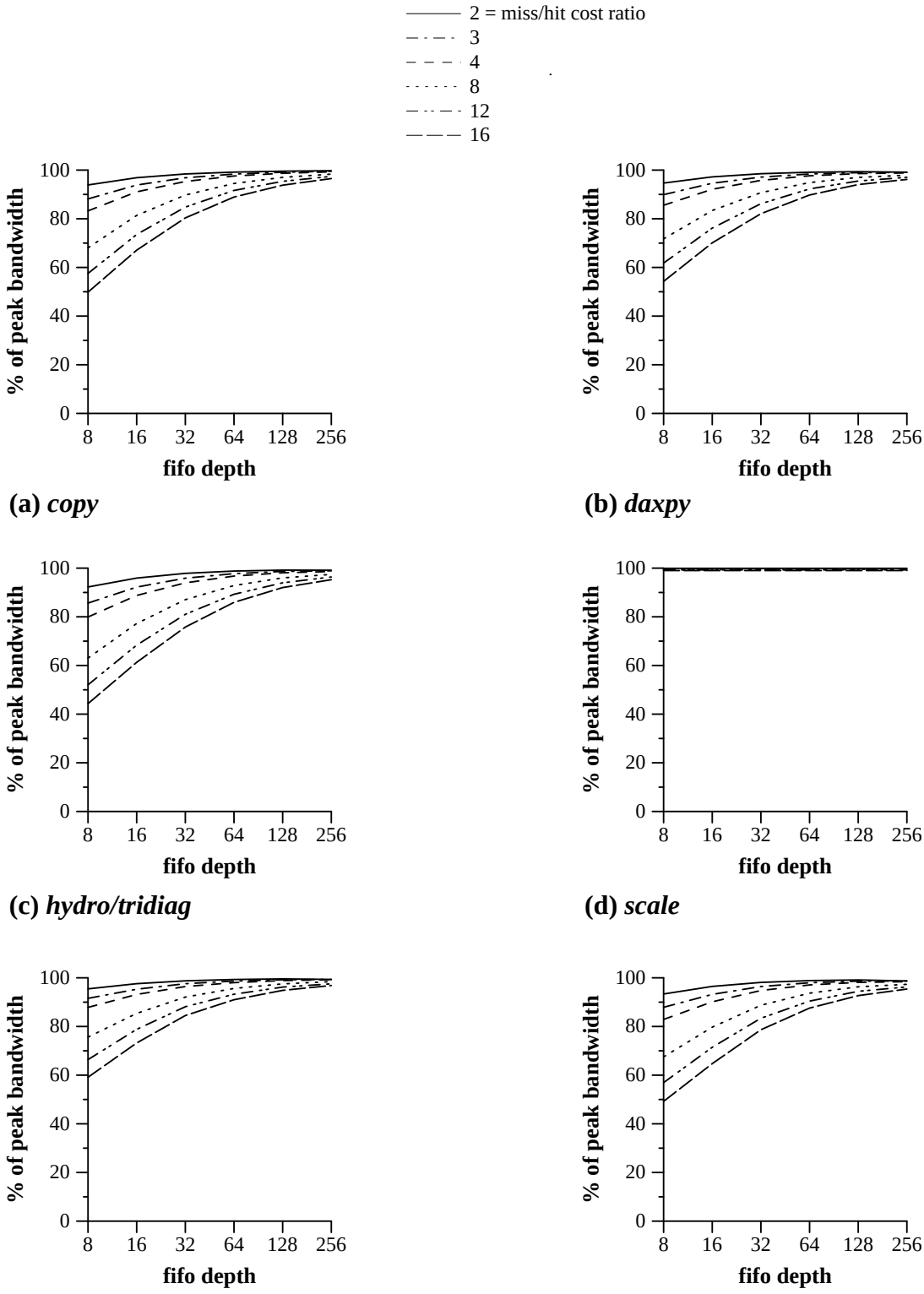
Of these algorithms, only T9 and T10 represent viable alternatives to the schemes explored in the last section. Algorithm T6 offers no real advantage — its performance is almost identical to T1’s. Algorithms T7 and T8 perform inconsistently in comparison to T1, sometimes yielding results several percentage points lower even for relatively deep FIFOs. They both perform better for the *hydro* benchmark and long vectors, but their performance on the other benchmarks, and even *hydro* with shorter vectors, is unpredictable, and unimpressive. Algorithm T9, on the other hand, only performs worse for very shallow FIFOs and memory systems with many banks. In general, its performance is competitive with the schemes from the previous section, although Algorithm T4 tends to perform slightly better in general for our benchmarks. Algorithm T10 performs much worse for shallow FIFOs and high interleaving factors, but if it were sufficiently cheap to implement, it might be a reasonable alternative, provided we could implement deep FIFOs.

7.3 Changing DRAM Page-Miss/Page-Hit Cost Ratios

Figure 58 through Figure 62 illustrate SMC performance for long vectors (10,000 elements) as the memory’s DRAM page-miss to page-hit cost ratio increases. As before, all performance curves are given as a percentage of peak bandwidth, thus for the systems with a miss/hit cost ratio of sixteen, it’s as if the page-misses required sixteen times as long to service. Figure 58 through Figure 61 may therefore appear a bit misleading, since the miss/hit ratio is likely to increase primarily as the result of a reduction of the page-hit time, rather than an increase in the page-miss time. At a ratio of sixteen the SMC is delivering a somewhat smaller percentage of a *much* larger available bandwidth — resulting in a significant net increase. To illustrate this, Figure 62 shows the performance of *hydro* for long vectors if we hold the page-miss cost constant and decrease the page-hit cost, increasing the total bandwidth proportionately.

If we hold the number of modules fixed and increase the page-miss/page-hit cost ratio, deeper FIFOs are required in order to amortize the page-miss costs. Note that relative performance is approximately constant if we scale FIFO depth linearly with miss/hit cost. The near-horizontal gray lines in Figure 61(a), Figure 61(c), and Figure 61(e) highlight this effect. Consider the *hydro* benchmark, for example. For an eight-bank memory with a miss/hit cost ratio of sixteen, an SMC with 256-deep FIFOs delivers 75.11% of peak bandwidth. With FIFOs that are 128 deep, the SMC achieves a similar performance — 75.93% — with a miss/hit cost ratio of eight. Likewise, when the miss/hit cost ratio is four and the FIFO depth is halved again, the SMC delivers 77.43% of peak bandwidth.

As we've seen before, as the interleaving factor grows, so must the FIFO depth. This is evident in the results of all benchmarks, including *scale*, which nonetheless achieves near-optimal bandwidth for all memory systems. Since this computation only involves one vector, every access after the first hits the current DRAM page. Performance is therefore invariant of the miss/hit cost ratio. For computations involving more than one vector, shallow buffers limit the number of page hits over which the SMC can amortize the cost of the inevitable page misses. *Scale* doesn't suffer from this, but its performance on the eight-bank memory system demonstrates another problem: with shallow FIFOs, the SMC cannot prefetch enough data to keep the processor from stalling. This inability to adequately overlap memory access with computation causes the benchmark to achieve over 20% less of the attainable bandwidth for eight- or sixteen-word buffers than it does for deeper FIFOs. Note that even the faster systems — those with a high interleaving factor or a high miss/hit cost ratio — still require only modest amounts of buffer storage.

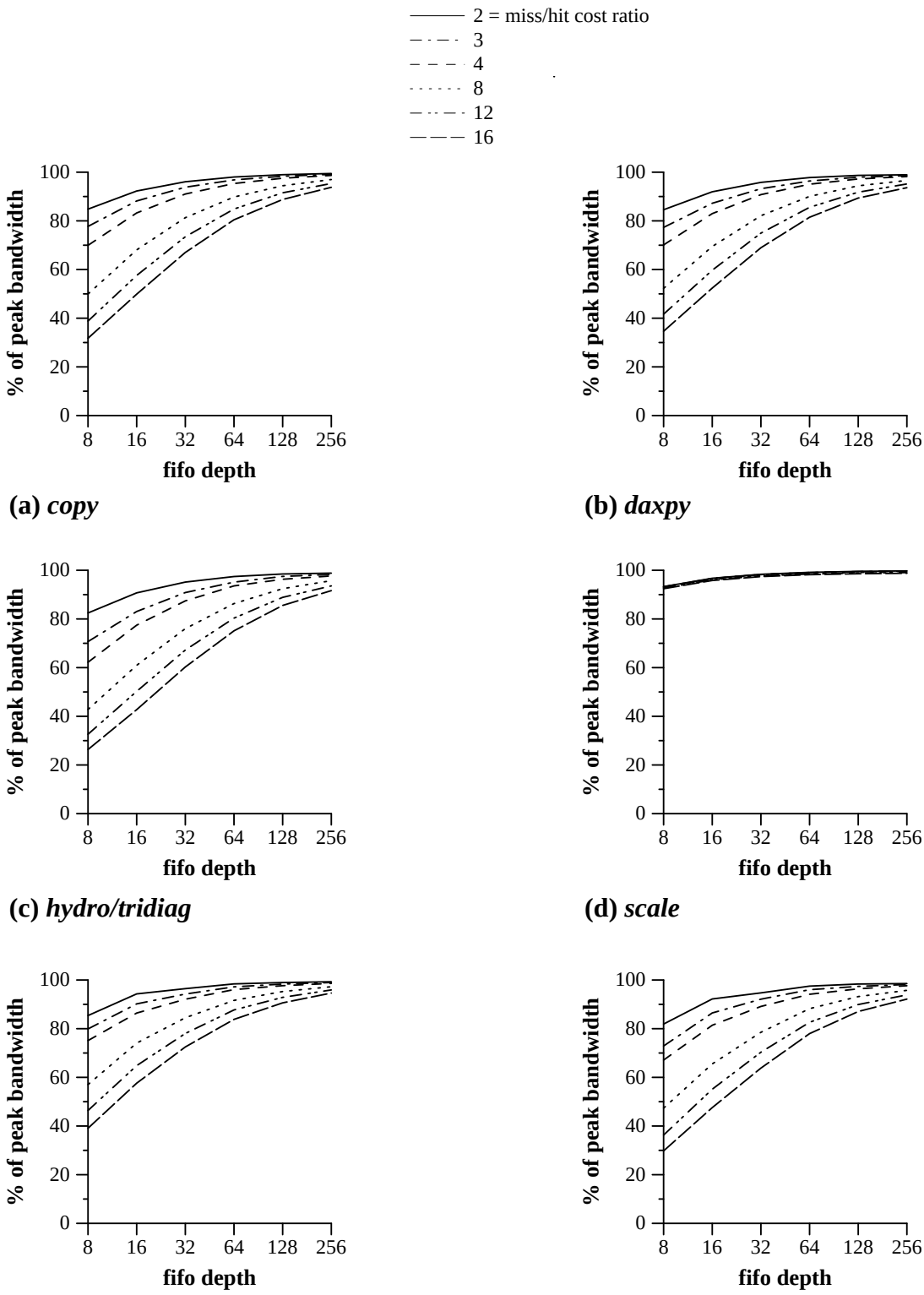


(a) ***copy*** **(b)** ***daxpy***

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

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

Figure 58 Varying Miss/Hit Cost Ratios on a Single-Bank Memory System

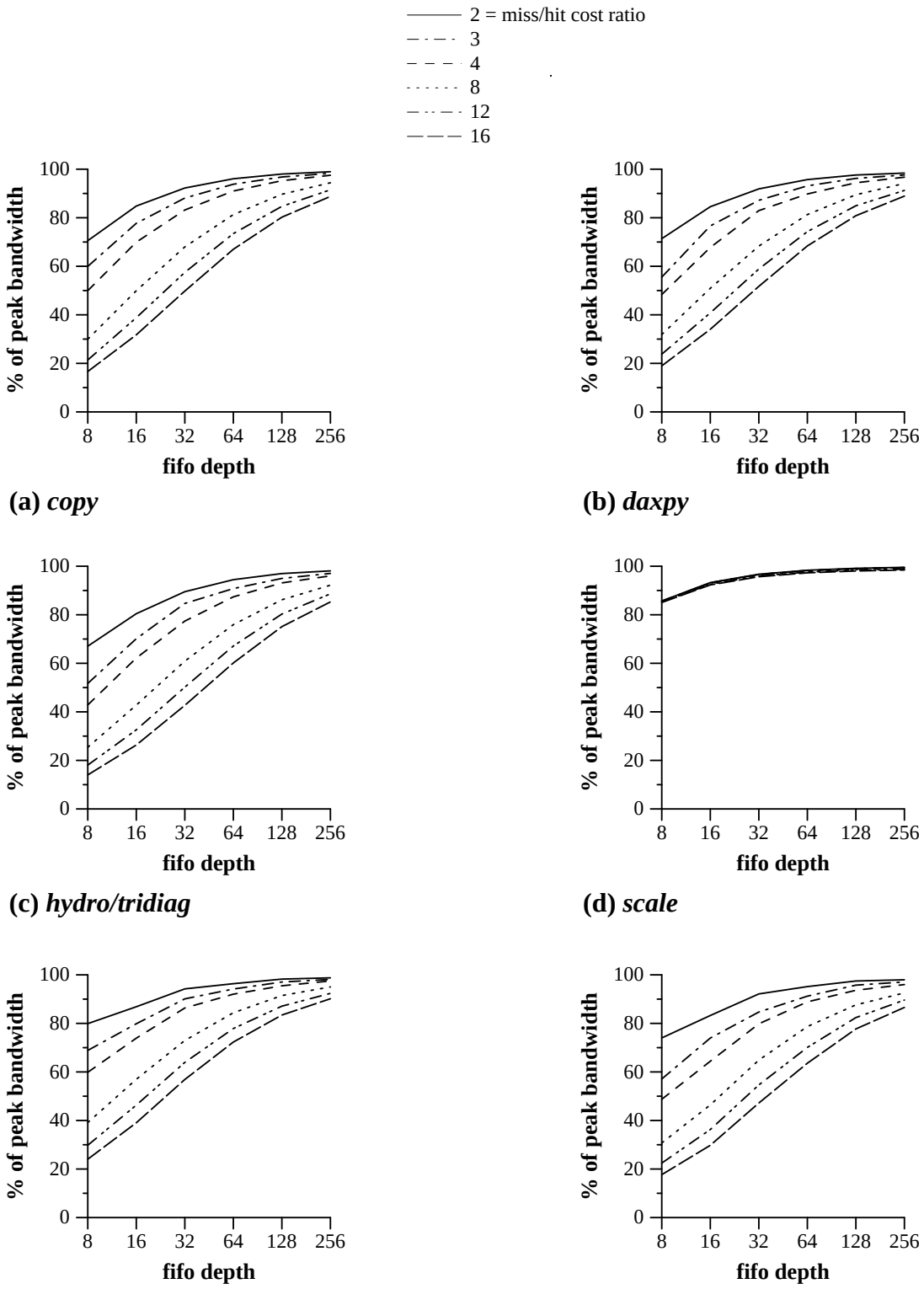


(a) ***copy*** **(b)** ***daxpy***

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

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

Figure 59 Varying Miss/Hit Cost Ratios on a Two-Bank Memory System



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

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

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

Figure 60 Varying Miss/Hit Cost Ratios on a Four-Bank Memory System

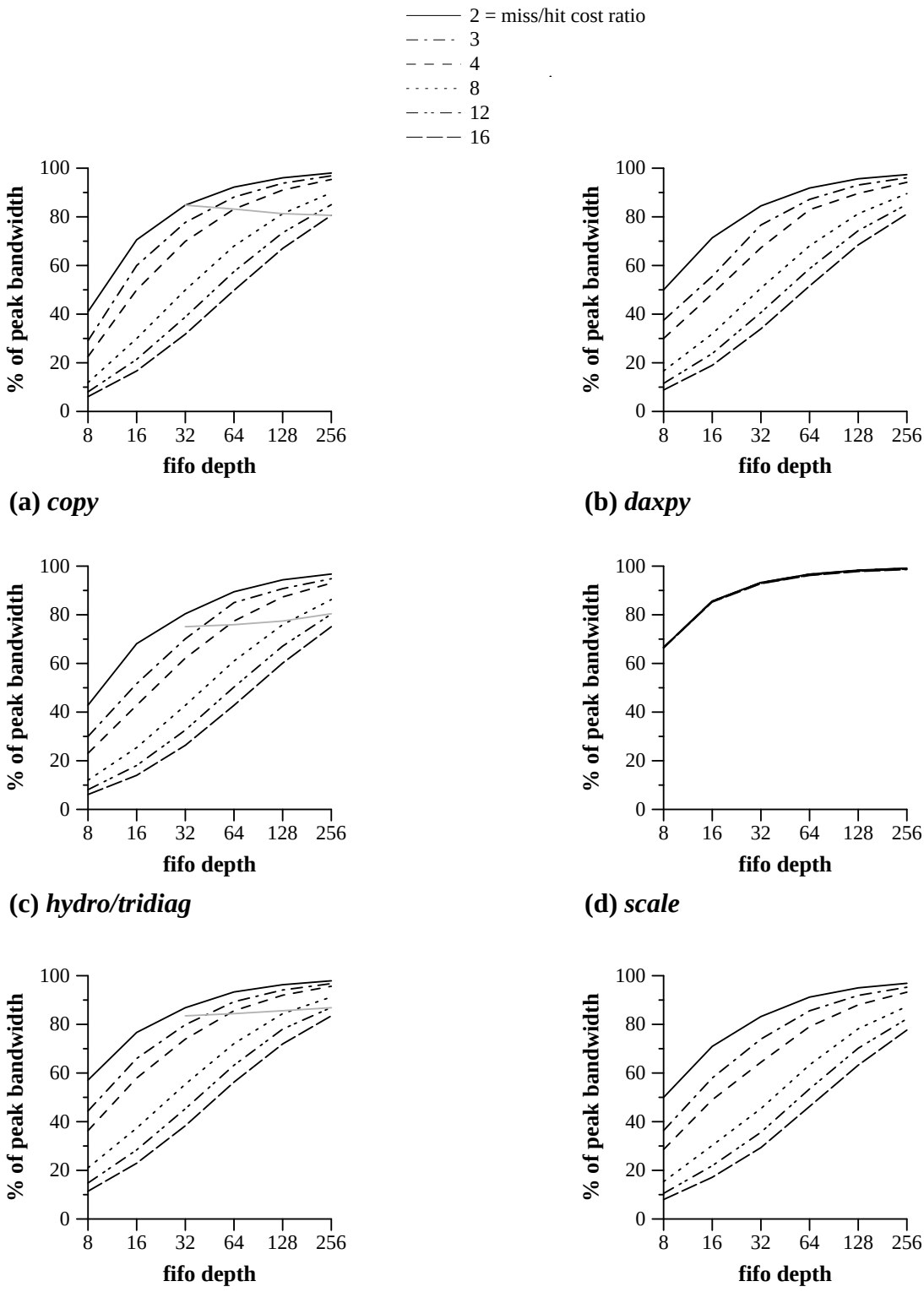


Figure 61 Varying Miss/Hit Cost Ratios on an Eight-Bank Memory System

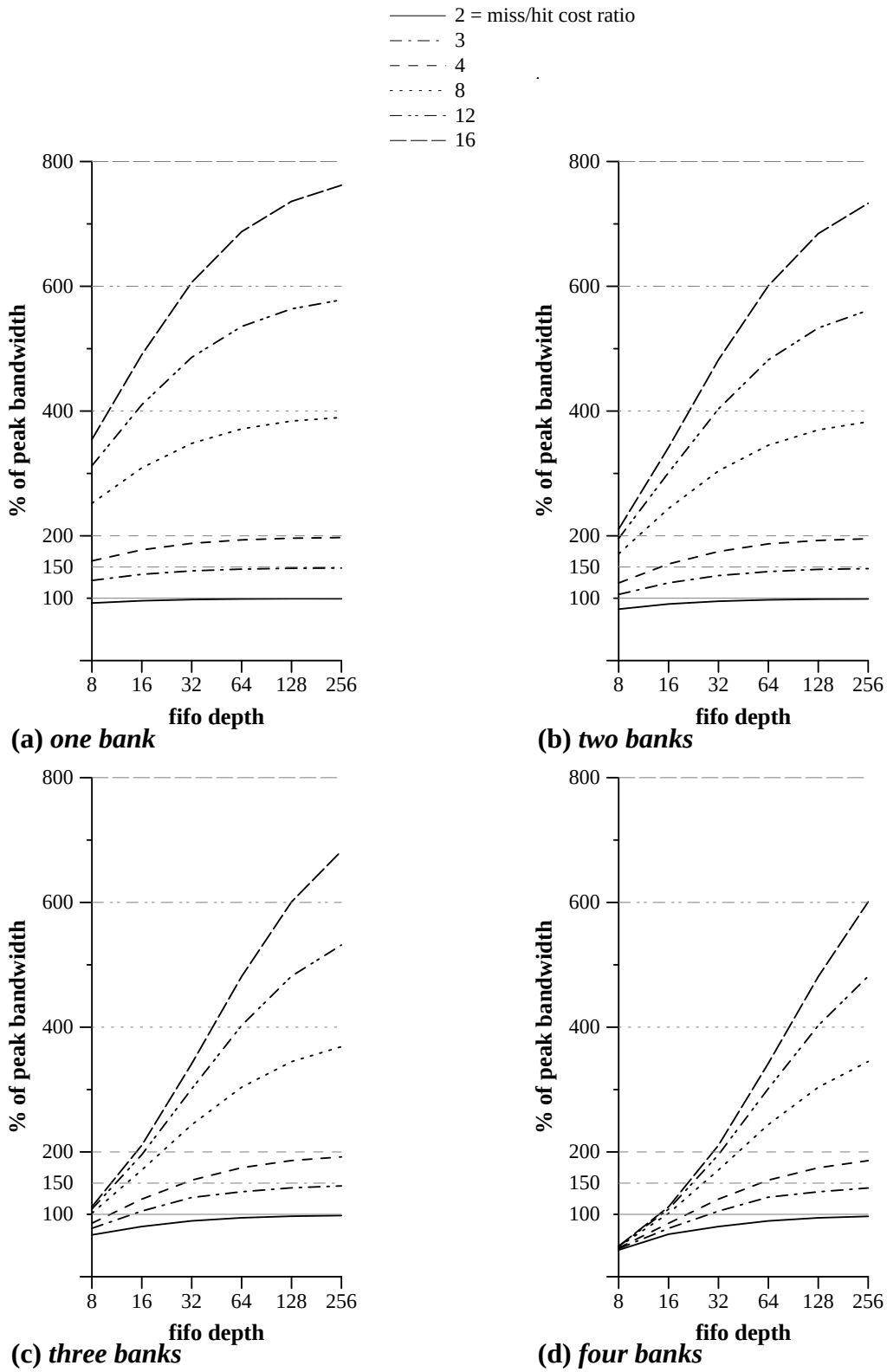


Figure 62 *hydro* — Varying Miss Costs on Four-Bank Memory System

7.4 Results Summary

The overwhelming similarity of the performance curves presented in Section 7.1 leads us to conclude that neither the ordering strategy nor the processor's access pattern has a large effect on the SMC's ability to optimize bandwidth. In fact, the simpler algorithms usually do as well or better than their more sophisticated counterparts. For the benchmarks and memory systems simulated, algorithms involving a "threshold of service" requirement behave inconsistently, and generally fail to outperform the simpler schemes.

Explicitly trying to take advantage of the memory system's available concurrency by initiating accesses in parallel (the P family of selection schemes) turns out to be of no real benefit, and occasionally hinders performance. Given that the SMC can only process one access at a time, it makes sense to initiate only one access each bus cycle. Performance between the "greedy round-robin" (R) scheme and the simpler "token passing" (T) scheme is sufficiently similar that deciding which is preferable becomes a question of implementation cost. The anitcipated additional complexity of implementing the former (R) scheme seems an unnecessary expense, as the latter (T) scheme should prove simpler and indeed faster.

Whether to choose T4 or T5 over A1 depends on the complexity of the circuitry required to implement each. The bank-centric schemes, T4 and T5, give better overall performance, but if A1 is sufficiently cheap to implement, the cost/performance tradeoffs might be worthwhile.

These simulation results reinforce the intuitive notion that FIFO depth *must* scale with the interleaving factor if we are to achieve good performance on a memory system with a large number of banks. Even the best ordering algorithms will fail to deliver good performance is stifled by inadequate buffer space. One final observation is that when faced with a choice

between implementing a more complicated (and better-performing) access ordering scheme and building deeper FIFOs, the latter will generally yield better performance.

8. Conclusions

Memory bandwidth is rapidly becoming the performance bottleneck in the application of high performance microprocessors to vector-like algorithms, including many of the “grand challenge” scientific problems. Caching is not the sole solution for these applications due to their poor temporal and spatial locality.

Achieving greater bandwidth requires exploiting the characteristics of the entire memory hierarchy; it cannot be treated as though it were uniform access-time RAM. Moreover, exploiting the memory’s properties will have to be done dynamically — essential information (such as alignment) will generally not be available at compile time.

Reordering 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.

Here we have reported the basic design of a uniprocessor Stream Memory Controller (SMC) and have analyzed its performance for a wide variety of design parameter values. Using current memory parts and only a few hundred words of buffer storage, an SMC system can consistently achieve nearly the peak bandwidth that the memory system can deliver. Moreover, it does so with naive code, and performance is independent of operand alignment.

Bibliography

[Bae91] Baer, J. L., and Chen, T. F., “An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty”, Supercomputing’91, November 1991.

[Bar92] Baron, R.L., and Higbie, L., *Computer Architecture*, Addison-Wesley, 1992.

[Ber89] Berry, M., et. al., “The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers”, CSRD Report No. 827, Center for Supercomputing Research and Development, Urbana, IL, May 1989.

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

[Cal91] Callahan, D., et. al., “Software Prefetching”, Fourth International Conference on Architectural Support for Programming Languages and 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.

[Dav90] Davidson, Jack W., and Benitez, Manuel E., “Code Generation for Streaming: An Access/Execute Mechanism”, Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, April 1991.

[DEC92] *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.

[Fu91] Fu, J. W. C., and Patel, J. H., “Data Prefetching in Multiprocessor Vector Cache Memories”, 18th International Symposium on Computer Architecture, May 1991.

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

[Goo85] Goodman, J. R., et al, “PIPE: A VLSI Decoupled Architecture”, Twelfth International Symposium on Computer Architecture, June 1985.

[Gup88] Gupta, R., Soffa, M., “Compile-time Techniques for Efficient Utilization of Parallel Memories”, SIGPLAN Not., 23, 9, 1988, pp. 235-246.

[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.

[Hay88] Hayes, J.P., *Computer Architecture and Organization*, McGraw-Hill, 1988.

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

[Hwa84] Hwang, K., and Briggs, F.A., *Computer Architecture and Parallel Processing*, McGraw-Hill, Inc., 1984.

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

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

[Jou90] Jouppi, N., “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers”, 17th International Symposium on Computer Architecture, May 1990.

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

[Kla91] Klaiber, A., et. al., “An Architecture for Software-Controlled Data Prefetching”, 18th International Symposium on Computer Architecture, May 1991.

[Lam91] Lam, Monica, et. al., “The Cache Performance and Optimizations of Blocked Algorithms”, Fourth International Conference on Architectural Support for Programming Languages and Systems, April 1991.

[Law79] Lawson, et. al., “Basic Linear Algebra Subprograms for Fortran Usage”, ACM Trans. Math. Soft., 5:3, 1979.

[Lee90] Lee, K., “On the Floating Point Performance of the i860 Microprocessor”, NAS Systems Division, NASA Ames Research Center, July 1990.

[Low93] Lowney, et. al., “The Multiflow Trace Scheduling Compiler”, Journal of Supercomputing, 7:1,2, May 1993.

[Mac93] Maccabe, A.B., *Computer Systems: Architecture, Organization, and Programming*, Richard D. Irwin, Inc., 1993.

[Man82] Mano, M.M., *Computer System Architecture*, 2nd ed., Prentice-Hall, Inc.,

1982

[McK93a] 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”, University of Virginia, TR CS-93-42, August 1993.

[McK93b] McKee, S.A., Moyer, S.A., Wulf, Wm.A., Hitchcock, C., “Increasing Memory Bandwidth for Vector Computations”, University of Virginia, TR CS-93-34, August 1993.

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

[Mea92] Meadows, L., Nakamoto, S., and Schuster, V., “A Vectorizing, Software Pipelining Compiler for LIW and Superscalar Architectures”, Proceedings of RISC’92, February 1992.

[Moy91] Moyer, S.A., “Performance of the iPSC/860 Node Architecture,” University of Virginia, IPC-TR-91-007, 1991.

[Moy92] Moyer, S., “Access-Ordering Algorithms for a Single Memory Module”, University of Virginia, IPC-TR-92-002, May 1992.

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

[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.

[Skl92] Sklenar, Ivan, “Prefetch Unit for Vector Operation on Scalar Computers”, Computer Architecture News, v 20, n 4, September 1992.

[Smi87] Smith, J. E., et al, “The ZS-1 Central Processor”, The Second International Conference on Architectural Support for Programming Languages and Systems, Oct. 1987

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

[Tom90] Tomek, I., *The Foundations of Computer Architecture and Organization*, Computer Science Press, 1990.

[Val91] Valero, M., et. al., “Increasing the Number of Strides for Conflict-Free Vector Access”, 19th International Symposium on Computer Architecture, May 1992.

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

[Wol89] Wolfe, M., “Optimizing Supercompilers for Supercomputers”, MIT Press, Cambridge, MA, 1989.

[Wul92] Wulf, W. A., “Evaluation of the WM Architecture”, 19th Annual International Symposium on Computer Architecture, May 1992.