

Instruction Cache Designs for a Class of Statically Scheduled Instruction Level Parallel Architectures *

Thomas M. Conte Sanjeev Banerjia Sergei Y. Larin Kishore N. Menezes
Sumedh W. Sathaye

Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC 27695-7911
E-mail: conte@eos.ncsu.edu

Abstract

Statically-scheduled architectures such as very long instruction word (VLIW) architectures use very wide instruction words in conjunction with high bandwidth to the instruction cache to achieve multiple instruction issue. The encoding used for the instructions can have an effect on the requirements placed on the instruction fetch and instruction cache hardware. One type of encoding is a *compressed encoding*, named so because it does not explicitly store NOPs within the wide instruction word. A compressed encoding enables high memory utilization but at the expense of variable-sized instructions and the complexities associated with fetching variable-sized instructions. This paper examines instruction fetch and instruction cache mechanisms for VLIW architectures that use compressed encodings. Relevant issues are investigated using the TINKER experimental testbed. A taxonomy for instruction caches for VLIW architectures that use a compressed encoding is introduced. Four cache organizations from different categories within the taxonomy are presented: the uncompressed cache, the banked cache, the rigid silo cache, and flexible silo cache. The designs are evaluated using trace-driven simulations. The results indicate that the banked cache is the best performer in terms of storage area requirements/program performance, and the silo cache designs could be appropriate in designs where storage limitations are not an issue or the characteristics of the applications to be executed are well-known.

1 Introduction

VLIW architectures use very wide instruction words to achieve multiple instruction issue. These architectures require high bandwidth instruction fetch (i-fetch) mechanisms to transport instruction words from the cache to the execution pipeline. In general, VLIW instructions are horizontally encoded wide words that issue an operation on every clock cycle to functional units (FUs) in the machine. The sequence of instruction words that compose a program is the *schedule* for the program. The instruction words can be encoded in several ways, and the choice of encoding can greatly influence the hardware required for i-fetch. A VLIW with an *uncompressed encoding* is one that explicitly stores NOP operations in the instruction word. The VLIW instruction stores a NOP in the operation slot for a particular FU if this FU is not scheduled to execute an operation at that point in the schedule. Use of an uncompressed encoding yields a fixed length instruction word, which can simplify the i-fetch hardware but at the expense of the potentially poor memory utilization.

Another class of encodings are *compressed encodings*, which do not store NOPs. VLIW instructions encoded using a compressed encoding are variably sized. The size of an instruction is dependent on the

*Review copy. Do not distribute.

number of FUs that will receive an operation at that point in the schedule. This type of encoding has a higher memory utilization and allows greater effective memory bandwidth than an uncompressed encoding. A compressed encoding also aids in object–code compatibility for VLIWs, such as in the dynamic rescheduling algorithm that has been proposed in the TINKER VLIW testbed [1]. A drawback is that such an encoding requires more complicated i-fetch support to handle the variable length instructions.

This paper focuses on the requirements of i-fetch imposed by a compressed encoding. Several mechanisms for i-fetch are presented, and the effect of each mechanism on instruction cache (i-cache) design is described. The paper is organized as follows: Section 2 introduces a basic instruction fetch mechanism and details the implementation of a compressed encoding; previous work in the area is also discussed. Section 3 presents a scheme for classifying i-caches for a compressed encoding and introduces four different organizations: the uncompressed cache, the banked cache, the rigid silo cache, and the flexible silo cache. Sections 4.1 through 4.3 describe each of the i-cache designs in detail and presents performance results from trace-driven simulations. Section 5 discusses the results, and Section 6 reviews related work. Section 7 concludes the paper.

2 A Basic Instruction Fetch Model

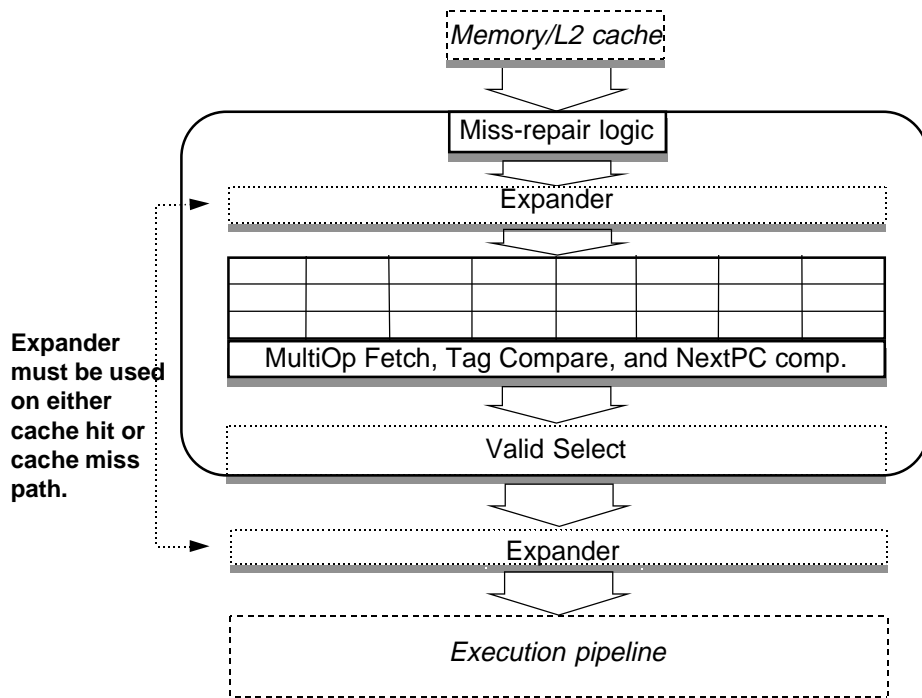


Figure 1: The basic instruction fetch model. All of the necessary stages for i-fetch for a compressed encoding are shown.

A simplified instruction fetch model for compressed encodings is shown in Figure 1. The solid lower borders in the diagram indicate pipeline latches that delineate the stages of the fetch pipeline. The main blocks in this model are the miss-repair logic, the pipelined instruction cache, and the expander, which is discussed below. The miss-repair logic handles cache miss repair requests, and could be implemented as a pipelined

interface to the next level in the memory hierarchy. Inside the instruction cache, each cache block holds one or more VLIW instructions. This is dependent on the instruction fetch mechanism and is explained in depth in Section 3. The cache consists of a cache fetch/tag compare stage which can be followed by optional valid select and expander stages. The valid select stage is required when multiple VLIWs can reside in a cache block and is used to select only the operations from the requested VLIW. For this study, it is assumed that all designs have the same cycle time.

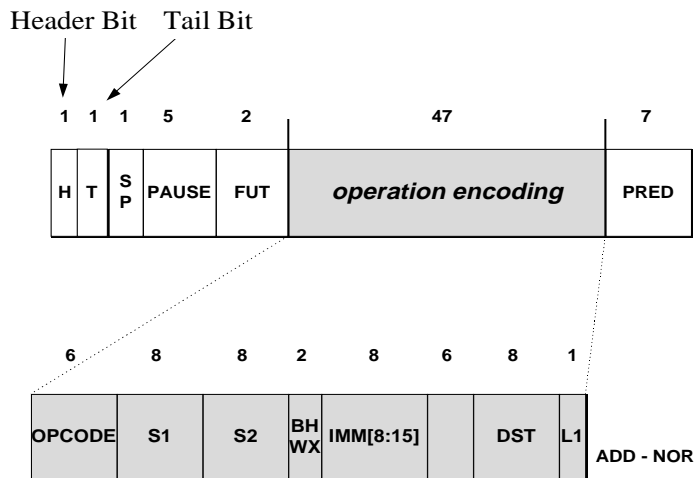


Figure 2: An integer add Op in the TINKER encoding.

A compressed instruction encoding using the TINKER VLIW experimental testbed is used for this study [2]. The compressed encoding combines individual operations (Ops) that can be issued in parallel into a unit of parallel issue called a MultiOp. A TINKER Op is a RISC-like instruction that is 64 bits in length. It can execute on one of four types of functional units (*FUType*): integer (integer computation and predicate handling), memory (loads and stores), FP (floating point add/mul/div/convert) and branch. An example of a TINKER integer add Op is shown in Figure 2. The TINKER encoding uses *header* and *tail* bits within an Op to delineate the beginning and end of a MultiOp, *i.e.*, the first Op in a MultiOp has its header bit set, and the last Op in a MultiOp has its tail bit set. The branch architecture is compiler-directed, similar to that of the PlayDoh specification from Hewlett Packard Laboratories [3]. For a n issue machine (TINKER- n), the maximum MultiOp size is $n * 64$ bits. A maximum-sized MultiOp contains an Op for each functional unit in the machine.

A useful property of compressed encodings is *rescheduling size invariance (RSI)*. RSI means that the size of a program does not vary across different generations of a VLIW architecture. Figure 3 shows that when using an uncompressed (non-RSI) encoding the size of an executable image can change. A change in code size can cause problems, such as branch target invalidation and constrained speculation [1]. A solution to the code size change problem is to use a RSI encoding like TINKER. The result of using the TINKER encoding to reschedule the code of Figure 3 is shown in Figure 4. Manipulation of the header and tail bits and the pause fields is the only requirement for modification of the schedule to execute correctly on different generations of an architecture¹.

Although a compressed encoding has several advantages, it also requires a complex i-fetch mechanism.

¹In general, the relative Op ordering may also change without ill effects, although this is not demonstrated in this example.

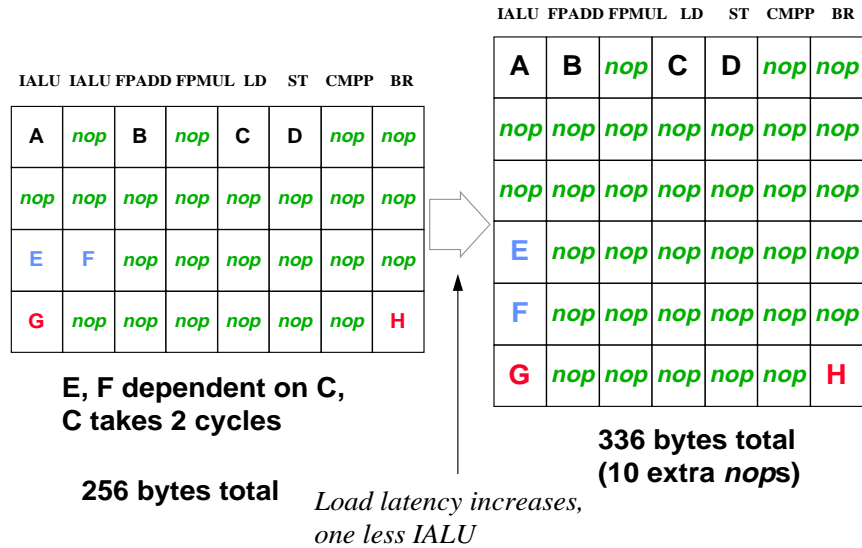


Figure 3: Object code compatibility using an uncompressed encoding. The size of the code changes when it is rescheduled for a different machine organization.

One step of i-fetch is NextPC generation, during which a PC is generated for the subsequent i-cache access. NextPC generation for an uncompressed encoding consists of adding a constant (the size of the fixed instruction) to the PC and using the new quantity (NextPC) to address the i-cache in the next cycle. Architectures that use variable length instructions have to first determine the length of the current instruction being fetched to determine what quantity to add to the PC to get NextPC. Another step of i-fetch is to determine how individual Ops in a MultiOp should be routed to FUs. For an uncompressed encoding, the FU is fixed by the position of the Op in the MultiOp, but this is not the case for a compressed encoding. A TINKER Op contains an FUT field indicating the functional unit on which it will execute. An Op destined for a particular type of functional unit can reside anywhere in a MultiOp, depending on the other types of Ops in the same MultiOp. This is intrinsic to a compressed encoding, and requires that Ops must be partially decoded and then *routed* to the appropriate functional unit. The *expander stage* performs this routing by routing all Ops in one MultiOp in parallel. An expander should have the functionality of a full crossbar; that is, route any type of Op to any FU (this requirement can be relaxed somewhat if the compiler enforces a partial ordering of Ops based on FUType).

An expander can be placed either on the cache hit path or the cache miss path as shown in Figure 1. A *miss path expander* is used only when a cache miss occurs and operates as follows: (1) As a MultiOp is fetched from memory, it is placed into the expander. (2) When the entire MultiOp has been received, the expander routes the Ops to specific positions in the cache, as selected by the miss address (i.e., the cache holds Ops in specific positions corresponding to their FUTypes).

Miss path expansion adds extra stages to the miss penalty. In contrast, a *hit path expander* is used on every cache access. After a MultiOp has been fetched from the cache, it is processed by the expander for Op-to-functional-unit routing. The expander is not on the cache miss path and therefore does not affect the miss penalty. However, the number of cycles needed for expansion is now in the fetch path and therefore adds to the branch misprediction penalty and the branch latency. For this reason, hit path expansion should

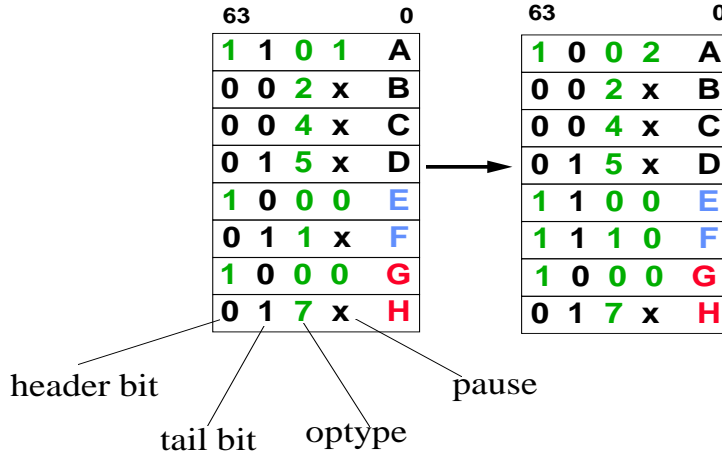


Figure 4: Object code compatibility using the TINKER compressed encoding. The size of the binary remains the same. The pause field for the first MultiOp is incremented by one, and the MultiOp originally consisting of Ops E and F has been made into two MultiOps.

be performed in one cycle, requiring a potentially complex and costly implementation.

3 A Classification for VLIW I-Caches

VLIW cache organizations can be classified based on two factors, the *degree of partitioning (DoP)* and the *NOPs policy* of the cache. The DoP describes the number of independent memory units (partitions) that are used to implement the cache and the placement of Ops into those memory units (an independent memory unit is a tag and data array that can be searched in parallel with other independent units). A traditional cache is a cache with one partition: all Ops can be stored at any location within the entire cache and only one tag or set of tags is searched². An alternative is a cache that uses multiple partitions, such that an Op maps into a particular partition based on its FUType. As described in Section 2, the TINKER encoding uses four FUTypes: integer (I), floating point (F), memory (M) and branch (B). A cache organization that assigns each FUType to a separate partition is represented as (I) (F) (M) (B) and is labeled *fully partitioned* (using this notation, a unified partition cache – a traditional cache – is represented as (I F M B)). Between the extremes of the unified partition and fully partitioned designs are *flexible* designs that permit multiple FUTypes to reside in a partition. For example, a design that allows integer and floating point Ops to share a partition and assigns Ops with other FUTypes to their own partitions is represented as (I F) (M) (B).

In implementation, a fully partitioned cache for a TINKER- n machine has n partitions, where every partition is the same size. A flexible partitioned cache combines the partitions for the sharing FUTypes and is the same size overall as a fully partitioned cache. Also, a flexible design allows the sharing FUTypes to reside at arbitrary locations within the combined partition.

The second factor for classification is whether the cache explicitly contains NOPs (a *NOP cache*) or not (a *NOPs-free cache*). The MultiOps held in a NOP cache are in uncompressed form, whereas in a NOPs-free cache they are in compressed form. The NOPs policy of a cache is closely tied to the placement of the expander in the i-cache pipeline (this is explained further in Section 4).

A permutation of the DoP and NOPs policies yields a wide variety of cache configurations. In this paper,

²Set associativity permits parallel tag compares, but this is orthogonal to partitioning.

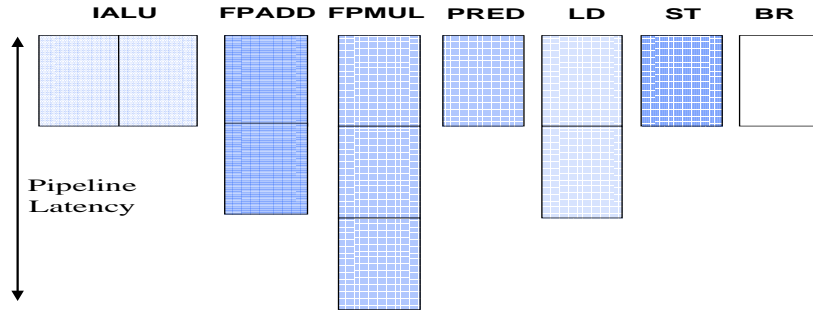


Figure 5: The TINKER-8 machine model.

Table 1: Benchmarks used for evaluation.

Benchmark Programs	
Integer	Floating point
085.gcc	039.wave5
124.m88ksim	048.ora
129.compress	052.alvinn
130.li	056.ear
134.perl	089.su2cor
147.vortex	090.hydro2d

four representative organizations are explored. The **uncompressed cache** has a DoP = (I F M B) (all Ops are stored in one partition) and a NOP policy. The **banked cache** has a DoP = (I F M B) (I F M B) and uses a NOPs-free policy. The **silos cache** has a DoP = (I) (F) (M) (B) (fully partitioned) and a NOP policy. Lastly, the **flexible silos cache** uses a combined partition for two FUTypes and separate partitions for the remaining two FUTypes, and a NOP policy. For the purpose of this study - an architectural evaluation - the four designs are assumed to have the same cycle time. Trace-driven simulations were used to evaluate the performance of the designs. The metric used for all simulations in this study is useful Ops completed Per Cycle (OPC). OPC captures the effect of i-cache performance on overall performance. The IMPACT compiler was used to superblock-schedule benchmark programs using the TINKER compressed encoding for the TINKER-8 machine organization [4]. The functional unit configuration and pipeline latencies are shown in Figure 5, with each functional unit pipelined to the depth indicated. A pipelined L1 cache/memory interface with a three cycle latency and a one Op bandwidth was assumed (the three cycle latency was chosen as it is similar to the L2 latency in contemporary microprocessors [5], [6]). A perfect L1 data cache and L2 cache was assumed to prevent data cache effects from coloring the performance measurements. Integer and floating point programs from the SPEC92 and SPEC95 suites were used as benchmarks for the evaluations and are listed in Table 1³. Two million Ops were sampled across the entire program for each simulation run.

³Results presented for floating point programs are lower than normal due to the lack of software pipelining of those programs.

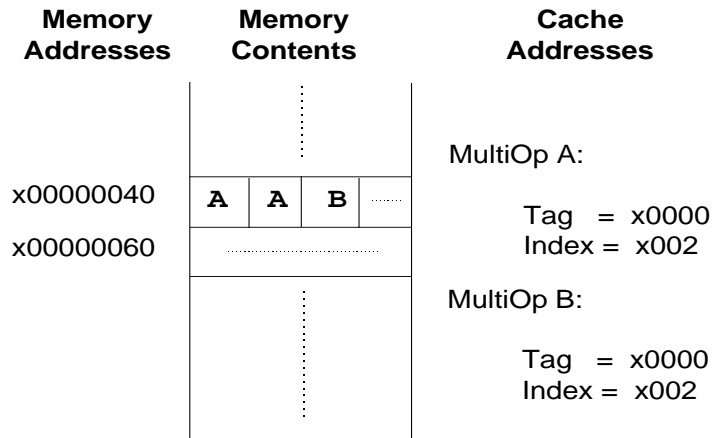


Figure 7: A traditional address mapping for the uncompressed cache.

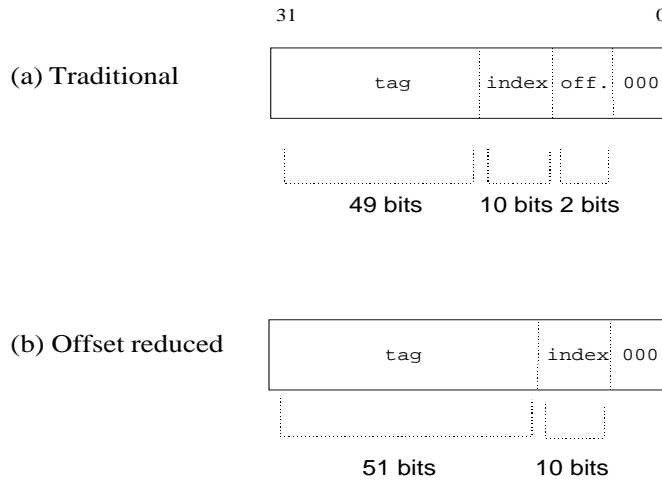
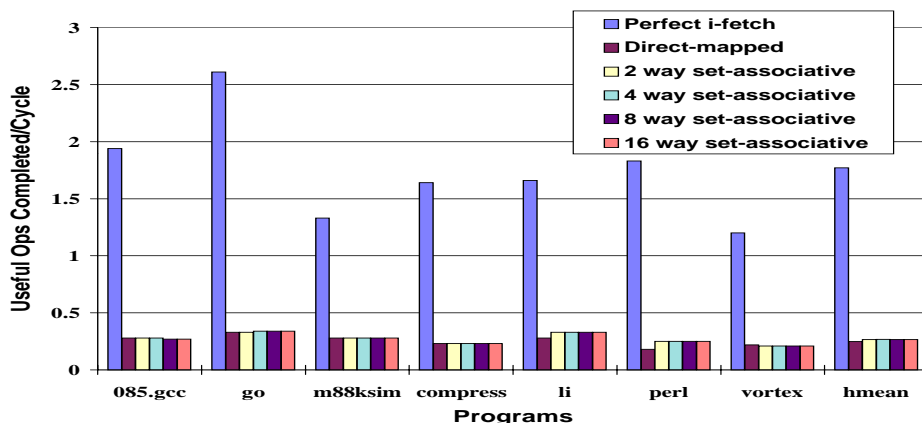


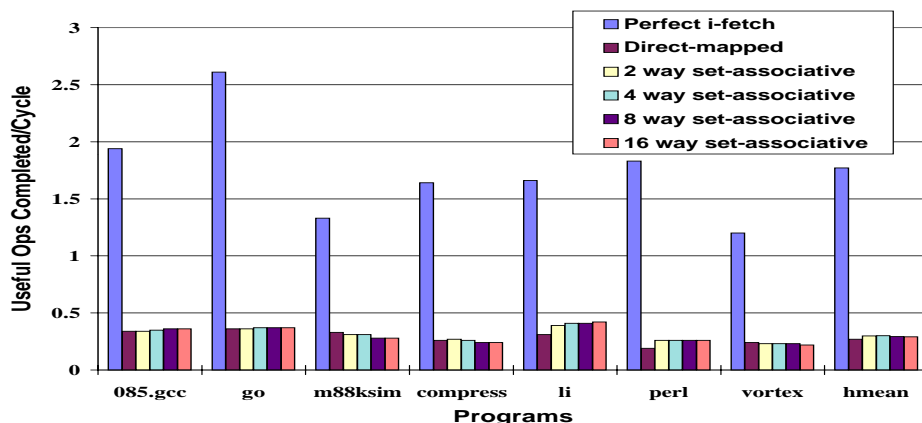
Figure 8: Interpreting a cache address – (a) traditional scheme, (b) offset-reduced scheme.

to be performed. The fetch hardware must add the length of the current MultiOp to the PC to generate the NextPC. Length computation can be performed at run-time, after the block containing the MultiOp is fetched from the cache. But this places NextPC computation *after* the block fetch, adding sequential computation at the end of the fetch cycle and possibly increasing cache access time. To avoid this situation, a MultiOp length field is associated with every cache block. The value for the length field is computed as Ops are received through the cache-memory interface. The length field is accessed in parallel with the tag and data, and the addition of the length field to the PC is done before the beginning of the tag compare stage. For a TINKER- n machine, a $\log_2 n$ -bit wide length field is needed for every cache block. For both 16KB and 32KB caches, this increases overall cache storage requirements by less than 0.5%.

The graphs in Figures 9 and 10 present the results of simulations that measured the performance of the uncompressed cache as compared to a perfect instruction fetch mechanism (a cache that never misses or a perfect i-cache). The uncompressed cache performs poorly when compared to a perfect i-cache. Although increasing the cache size from 16 KB to 32 KB yields better performance, the resulting OPCs are still



a) 16KB uncompressed cache, integer programs

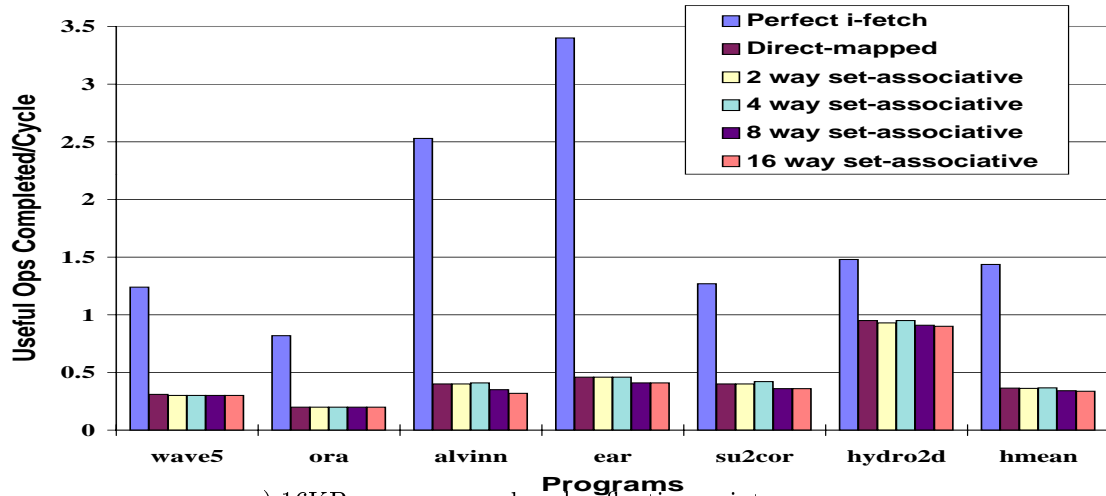


b) 32KB uncompressed cache, integer programs

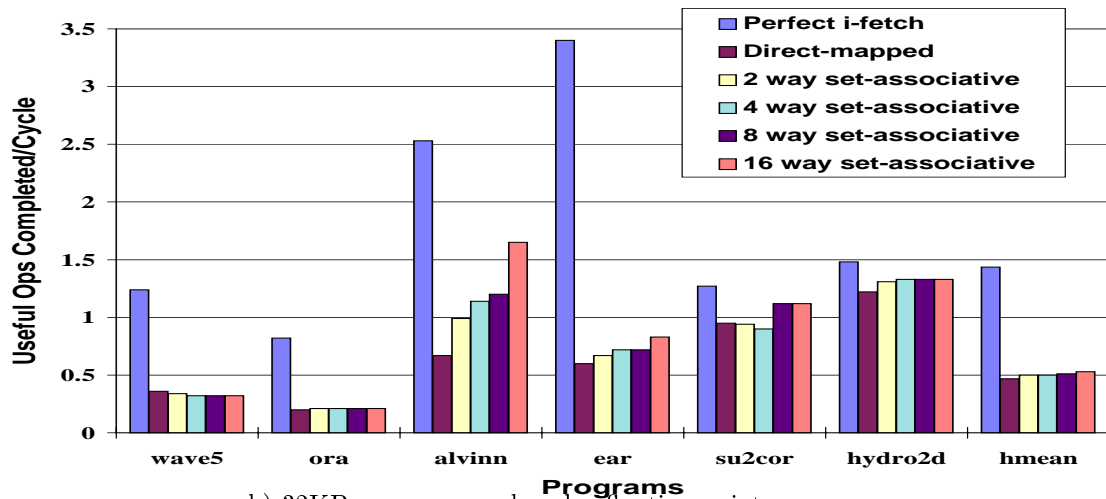
Figure 9: Performance of the uncompressed cache on integer benchmarks. Figure a) shows the performance for a 16KB cache, and Figure b) shows the performance for a 32KB cache. The metric used is useful Ops completed per cycle (OPC). The set of bars on the far right of each graph is harmonic means.

much smaller than the perfect cache. For the integer benchmarks, increasing the cache size yields 22% better performance but is still more than five times worse than a perfect i-cache. For the floating point benchmarks, the larger cache performed 19% better but is still four times worse than the perfect i-cache.

Increasing cache associativity did not have a consistent effect on performance. This indicates that the benchmark programs did not incur many conflict misses (which could be alleviated using associativity) but did incur a large number of capacity misses (which can be alleviated using larger caches). In many cases, an increase in associativity had a negative effect. This might seem counter-intuitive at first, but the reason is actually straightforward. When increasing cache associativity, the number of locations into which a MultiOp can map (the *height* of the cache) is reduced, since the number of index bits is reduced. This causes many MultiOps to map into the same cache indices. Normally, associativity eliminates misses caused by these cache conflicts. For an uncompressed cache, however, many of the benchmark programs map into a very small set of indices, and the number of conflicts within these indices is not mitigated adequately by the increased associativity. Hence, performance degrades slightly. This effect is more apparent on the floating



a) 16KB uncompressed cache, floating point programs



b) 32KB uncompressed cache, floating point programs

Figure 10: Performance of the uncompressed cache on floating point benchmarks. Figure a) shows the performance for a 16KB cache, and Figure b) shows the performance for a 32KB cache. The metric used is useful Ops completed per cycle (OPC). The set of bars on the far right of each graph is harmonic means.

point programs than on the integer programs, especially for the 16KB cache. However, associativity did yield benefits for some combinations of program and cache size. Several floating point programs - ear, su2cor, and hydro2d - show improved OPCs when associativity was increased for a 32KB cache. The program alvinn showed a marked increase in performance for higher associativities with a 32KB cache (as much as 90% for a 16 way set-associative cache compared to a direct-mapped cache configuration). For the integer programs, li was the only program that experienced a consistent performance increase with higher degrees of associativity.

The reason for the poor performance of the uncompressed cache is the extremely low space utilization of the uncompressed design. The maximum OPC of < 2 for the perfect cache indicates that only two out of eight words in a cache block are typically used. Possible solutions are to use a larger cache or a design that can place more than one MultiOp in a cache block. The latter is explored in the next section.

4.2 The Banked Cache

Figure 11 shows the organization of the instruction fetch mechanism when using a *banked cache*. The cache is organized as two data and tag arrays, as in the Intel Pentium processor [7]. The cache block size is the same as the machine width. MultiOps in the cache do not contain NOPs, and more than one MultiOp can reside in a cache block. A MultiOp can span two cache blocks. For this reason, on every clock cycle, two cache blocks are accessed: the block in which the requested MultiOp could reside (the *current block*) and the next sequential block (the *successor block*). In contrast to the uncompressed cache, the expander is on the cache hit path, and a valid select stage is also required. The extra stages increase the branch misprediction penalty to three cycles. The cache can be organized either as a direct-mapped cache or as a set-associative cache with LRU replacement.

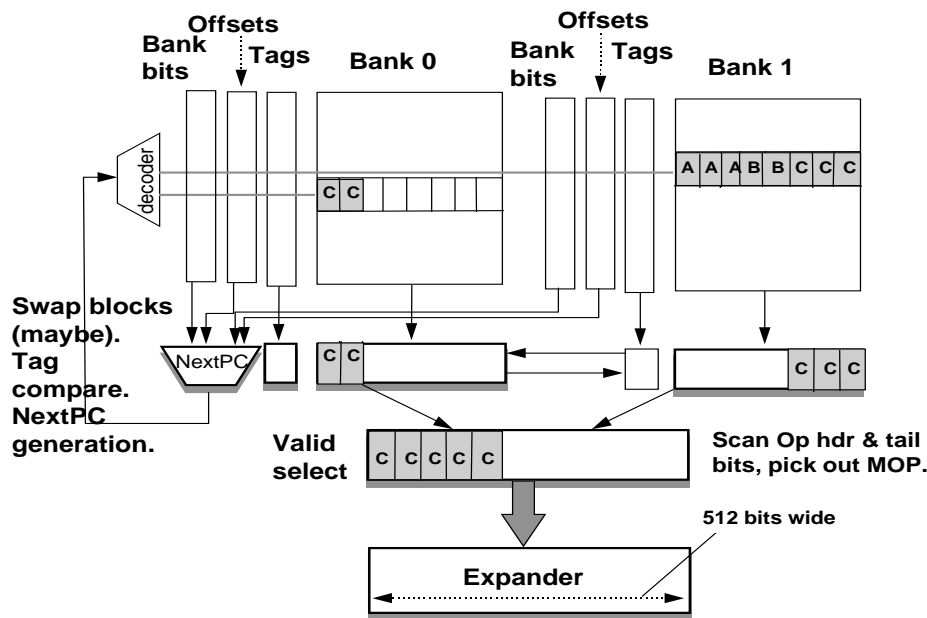


Figure 11: The banked cache. A fetch for MultiOp C is shown. Blocks from both banks are fetched and then swapped (if needed) before being passed to the single-cycle expander. NextPC computation is performed in parallel with cache access.

Addressing in the banked cache is similar to a traditional cache. The high order bits of the address are used as a tag, and the middle bits are used as an index. Because a MultiOp is variable length and does not always begin on a n -word boundary in cache, the low order bits are used as an offset to index to the start of the MultiOp in the cache block. When a PC is presented to the cache, an address decoder selects consecutive blocks in both cache banks.

The fetch mechanism is shown in Figure 11 with an example fetch operation. The first three Ops of MultiOp C occupy the last three Ops of the cache block in bank one, and the last two Ops of C occupy the first two Ops of the next cache block in bank zero. The PC is the address of the first Op of C at the beginning of the fetch cycle. There are three sequential steps required to fetch MultiOp C. These are shown in Figure 11 and detailed below:

1. The current block containing the first three Ops and the successor block containing the fourth and fifth Ops are requested from the cache.

2. For correct alignment of the MultiOp, the fetch hardware must know where the last Op of MultiOp C lies in the successor cache block. To do this, it searches for the tail bit of the last Op in C. This information permits the cache fetch stage to perform correct alignment by swapping the two MultiOp fragments. Note that a banking factor of two facilitates such an exchange.
3. The header bits for all Ops in the blocks are scanned to determine the Ops belonging to MultiOp C, starting from the location of the first Op in the requested MultiOp. Valid select lines are then enabled to pass only the requested Ops to the expander stage.

While fetching the current MultiOp, NextPC must also be computed. In the absence of an Op that changes the control flow of the program, this is accomplished by using extra bits to store offset information for MultiOps in the cache. Details of the hardware are pictured in Figure 12. An offset field and a bank bit are maintained for every Op in a cache block. A valid bit is associated with every offset field. The offset field for a MultiOp M_i indicates the offset within the cache block of the next sequential MultiOp M_{i+1} and is set only for the first Op in M_i . The bank bit indicates whether M_{i+1} resides in the same bank as M_i or in the next one. The values for these fields are set as Ops are received in the cache-memory interface at cache miss time so no extra cycles are required.

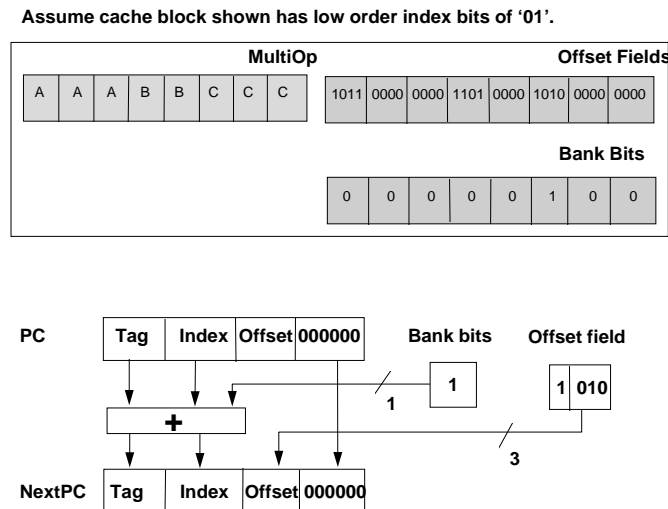
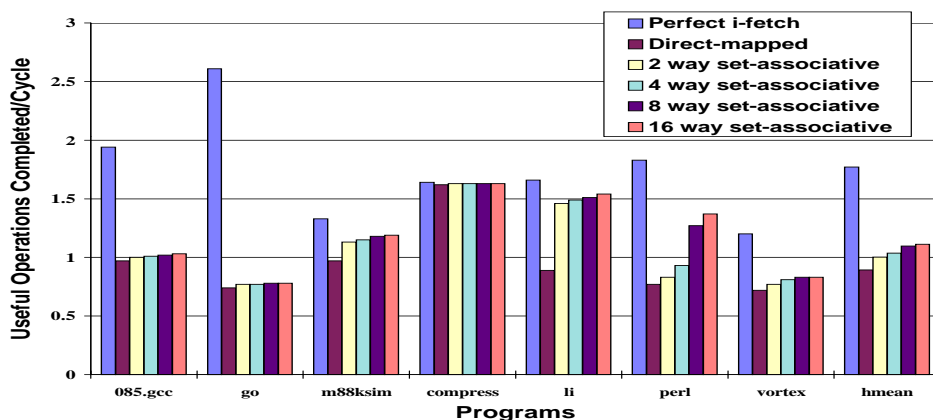


Figure 12: NextPC computation for the banked cache. A cache access for MultiOp C is shown. The offset field for C is placed into the appropriate positions and the bank bit is added to the remaining high order bits to form a new index and tag.

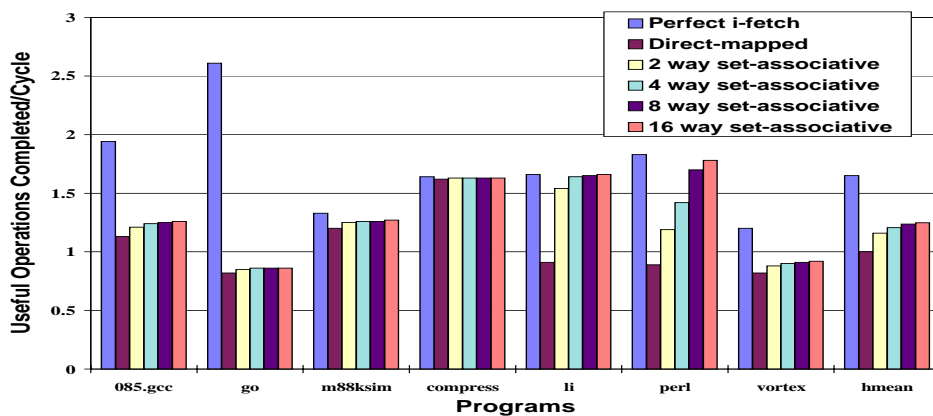
The steps for NextPC computation are as follows. On a cache miss for MultiOp M_i , Op fetches are generated through a pipelined memory interface to the next level of the memory hierarchy. Since the size of the MultiOp is unknown at miss time, enough Op fetches are generated to guarantee retrieval of the largest possible MultiOp. The unit of replacement is a cache block, so the number of fetches generated is always a multiple of the block size. If the miss address is on a cache block boundary (offset 0), n Op fetches are generated. Otherwise, $2n$ fetches are generated, since M_i could span two blocks. (Note that the latter case prefetches at least some of the Ops for M_{i+1} . This is termed *implicit prefetching*.) As the Ops are received through the cache-memory interface, the boundaries between MultiOps are detected by examining the header and tail bits of each Op, and a counter counts the number of Ops in each MultiOp. The length

l_i of M_i is combined with its starting offset o_i to determine the starting offset o_{i+1} of M_{i+1} . If $o_i < o_{i+1}$, M_i and M_{i+1} begin in the same cache bank, and M_i 's bank bit is set to 0. Otherwise, the MultiOps begin in different banks, and M_i 's bank bit is set to 1. The offset field and the bank bit are accessed in parallel with the tag and data arrays during cache access. Details on how the fields are used and the combinational logic required for NextPC computation are detailed in a technical report [8].

The use of dedicated storage within the cache to aid in NextPC computation is similar to the use of successor indices initially proposed by Johnson [9] and implemented in the AMD K5 [10], although that scheme is for an unbanked cache. For a TINKER- n banked cache, $\log_2 n$ offset bits per word (Op) are needed, plus a valid bit and a bank bit. For 16KB and 32KB caches, this increases overall cache storage requirements by approximately 7.1%.



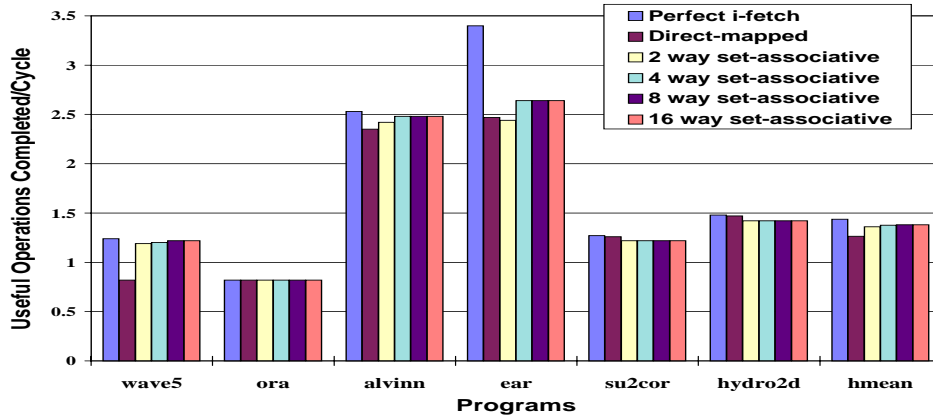
a) 16KB banked cache, integer programs



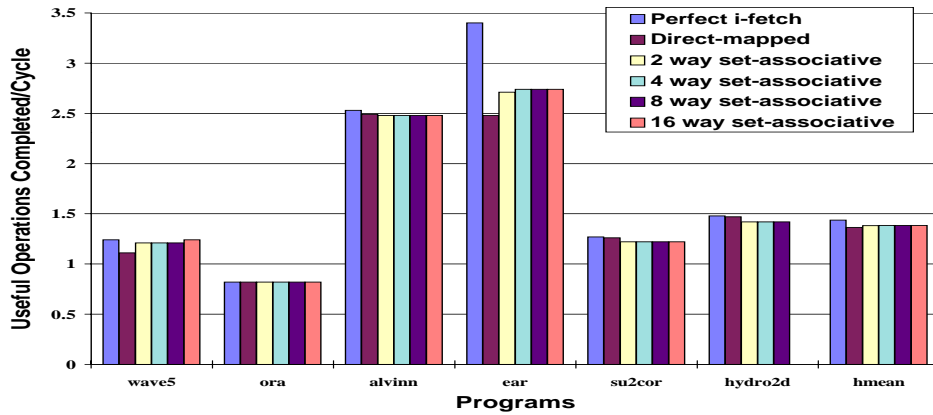
b) 32KB banked cache, integer programs

Figure 13: Performance of the banked cache on integer benchmarks. Figure a) shows the performance for a 16KB cache, and Figure b) shows the performance for a 32KB cache. The metric used is useful Ops completed per cycle (OPC). The set of bars on the far right of each graph is harmonic means.

The graphs in Figures 13 and 14 present the results of simulations that measured the performance of the banked cache against a perfect i-fetch mechanism. The banked cache performs over an order of magnitude better than the uncompressed cache. The greater utilization of the data storage area offsets the greater



a) 16KB banked cache, floating point programs



b) 32KB banked cache, floating point programs

Figure 14: Performance of the banked cache on floating point benchmarks. Figure a) shows the performance for a 16KB cache, and Figure b) shows the performance for a 32KB cache. The metric used is useful Ops completed per cycle (OPC). The set of bars on the far right of each graph is harmonic means.

branch misprediction penalty. Increasing the cache size from 16KB to 32KB boosts the performance of the integer programs by 12 % and the performance of the floating point programs by 8 %, for the direct-mapped configurations. The compress program fits almost entirely within the 16KB cache, and m88ksim fits almost entirely within the 32KB cache. For the floating point programs, the 16KB banked cache is large enough to hold nearly all of four of the six programs, and the 32KB banked cache is large enough to nearly hold all of five of the six programs.

The use of associativity also has a strong positive effect, especially for the integer programs. Associativity has a particularly dramatic effect on perl and li. Their OPCs increase by 100% and 81%, respectively, when associativity is increased from 1 way (direct-mapped) to 16 way for the 32KB cache. When associativity is increased by a lesser degree (1 way to 2 way), their OPCs still increase by 33% and 69%, respectively. Associativity does not have as strong an influence on the integer programs that have larger cache footprints - 085.gcc, go, and vortex. These programs benefit more from increasing the cache size from 16KB to 32KB and suffer more from capacity misses than from conflict misses. Associativity also helps the floating point

programs, but the results are less consistent. Many of the programs are nearly cache resident in the direct-mapped designs, so the potential benefits to be gained from associativity are minimal. Adding associativity yields small improvements for *alvinn* and *ear* and actually degrades performance slightly for *su2cor* and *hydro2d* (for the same reasons as explained in Section 4.1). *wave5* is the exception among the floating point programs; it’s performance improves by 45% and 9% when going from 1 way to 2 way associativity for the 16KB and 32KB caches, respectively.

The banked cache is clearly a much better performer than the uncompressed cache. However, the performance is still lower than the perfect cache, by as much as 25%. Some of this penalty may be due to overfetching an entire cache block when only a portion of the block is requested. This hypothesis is tested in the next section.

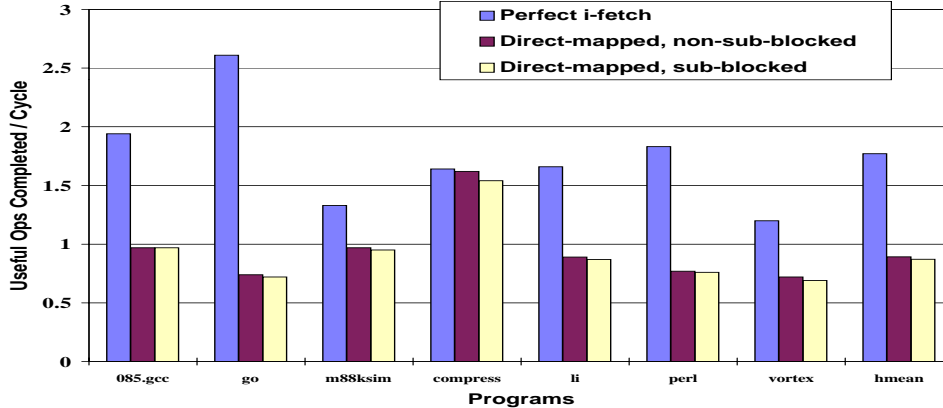
4.2.1 Sub-blocking in the banked cache

A variant of the banked cache is to use *sub-blocking* [11]. Sub-blocking partitions each cache block into smaller units (sub-blocks). Sub-blocking usually increases the miss ratio but reduces the amount of memory traffic as each sub-block can be independently filled and replaced. For miss repair with sub-blocking, Ops are retrieved starting from the address of the beginning of the MultiOp, not the address of the beginning of the cache block, as is done without sub-blocking (this is called *block fetching*). The advantage is reduction of the time for miss repair, at the expense of a small increase in the miss ratio because instructions preceding the requested instruction are not filled in the cache block. When a miss occurs, the length of the missing MultiOp is not known. Unless the miss occurs on a cache block boundary, enough memory fetches are generated to fill to the end of the successor block (this is required because a MultiOp can span two blocks). If the MultiOp resides entirely within the current block, the successor block is not filled. A valid bit is associated with each sub-block and is set when the sub-block is filled (sub-block valid bits for all preceding Ops in the current block are unset). Offset fields and bank bits can be used for NextPC computation. A sub-blocked cache can employ a form of prefetching termed *load forwarding* [12]. Load forwarding for an instruction cache consists of fetching the sub-block in which the requested MultiOp M lies and all subsequent sub-blocks to the end of the cache block.

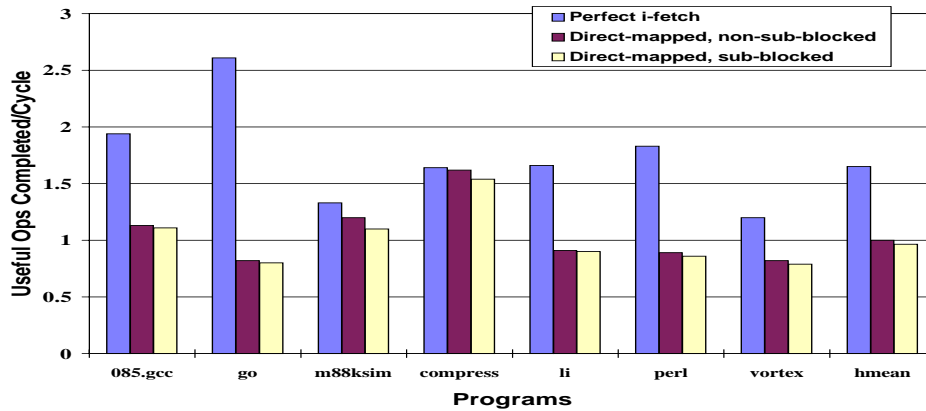
A direct-mapped banked i-cache with one Op-sized sub-blocks was simulated, and the results are presented in Figure 16. Overall, the sub-blocking designs performed slightly worse than the designs without sub-blocking. Further examination of the behavior of the programs revealed why. Much of the code involved backward branches that branched to the beginning of cache blocks. Although the latter part of the block had been filled during an earlier demand request, the lower offsets had not. Contrast this with the block fetch design in which the entire block is filled when a miss occurs. The sub-blocked designs therefore generated memory requests in situations where the block fetch designs did not. Overall, the difference in performance between the sub-blocked and non-sub-blocked designs was quite small, less than 1%.

4.3 The Silo Cache

The silo cache is organized as a series of partitions or *silos*, where each silo holds Ops for a particular FU or set of FUs, as shown in Figure 17. The DoP can range from a unified partition to one-FUType-per-partition. Each entry in a silo can hold one Op, and has associated with it a tag and a length field and a valid bit for NextPC computation. The silos can be organized in a direct-mapped or set-associative manner. When set-associativity is used, not only are the silos independently searched, but within each silo, parallel tag compares are performed among individual Op entries.



a) 16KB banked sub-blocked cache, integer programs



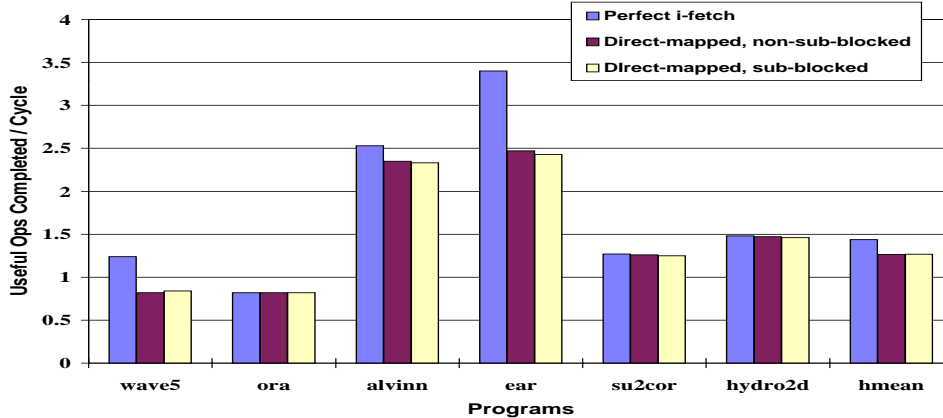
b) 32KB banked sub-blocked cache, integer programs

Figure 15: Performance of the banked sub-blocked cache on integer benchmarks. Figure a) shows the performance for a 16KB cache, and Figure b) shows the performance for a 32KB cache. The metric used is useful Ops completed per cycle (OPC). The set of bars on the far right of each graph is harmonic means.

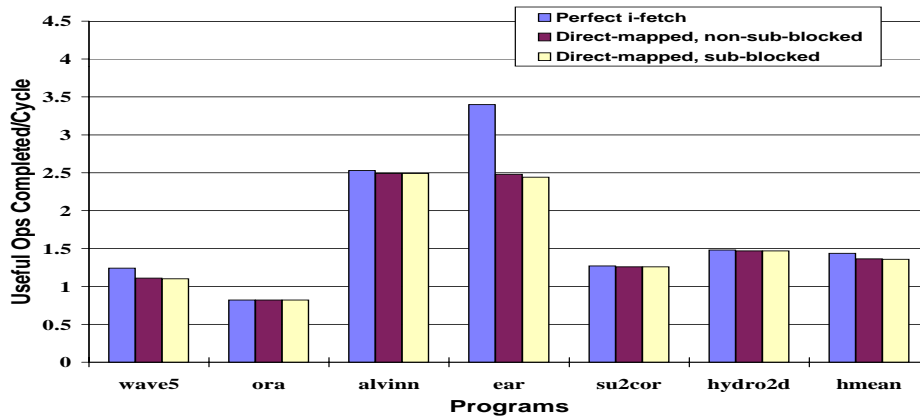
4.3.1 The Rigid Silo Cache

The *rigid silo cache* is pictured in Figure 17. It is a fully partitioned cache, with a degree of partitioning of (I) (F) (M) (B). The cache block size is the same as the machine width. The design uses a miss path expander (two cycle expansion is assumed). The branch misprediction penalty is one cycle.

As with the uncompressed cache, in a silo cache sequential MultiOps can map to the same cache block and cause conflict misses if the address is interpreted in a traditional manner, as shown in Figure 8(a). For this reason, the silo cache uses the offset-reduced scheme to interpret the address (see Section 4.1). The index bits are used to address the silos and map to the same location in each silo. Tag comparisons for all silos are done in parallel. A hit is signaled by a tag match and a valid length field (as shown for MultiOp B in Figure 17, the tags and length bits for all Ops in a MultiOp are identical). For a cache hit, the Ops for the MultiOp are directed to the functional units associated with their specific silos. The silos can be organized in a direct-mapped or set-associative manner. LRU replacement is used for set-associative silos.



a) 16KB banked sub-blocked cache, floating point programs



a) 32KB banked sub-blocked cache, floating point programs

Figure 16: Performance of the banked sub-blocked cache on floating point benchmarks. Figure a) shows the performance for a 16KB cache, and Figure b) shows the performance for a 32KB cache. The metric used is useful Ops completed per cycle (OPC). The set of bars on the far right of each graph is harmonic means.

A miss occurs when either none of the tags in any of the silos match or a length field in any silo is invalid. A fetch for the missing MultiOp is then initiated to the next level of the memory hierarchy. After the MultiOp has been fetched into the cache–memory interface, it is expanded, and each Op is routed to an appropriate silo. Values for the length fields are computed in parallel with expansion. For each silo that receives an Op, the corresponding tag, length field, and valid bit are updated. A situation can occur where some but not all of the Ops for a MultiOp are replaced during a cache fill. The Ops that are not replaced are flagged to indicate that the entire “parent” MultiOp is no longer cache-resident. This is accomplished by searching for all Ops that have the same tags as the Ops being replaced and clearing their valid bits. The next cache access for the partially displaced MultiOp results in a miss, due to the reset valid bits.

During a cache fill, if a particular silo does not have an Op routed to it by the expander, the Op position corresponding to the fill address in that silo is left empty. In this respect, the silo cache uses a NOP policy. However, several MultiOps can reside at the same address across all silos, which is a hybrid between the NOP and NOPs-free policies. To understand this, note that each silo is addressed individually and performs

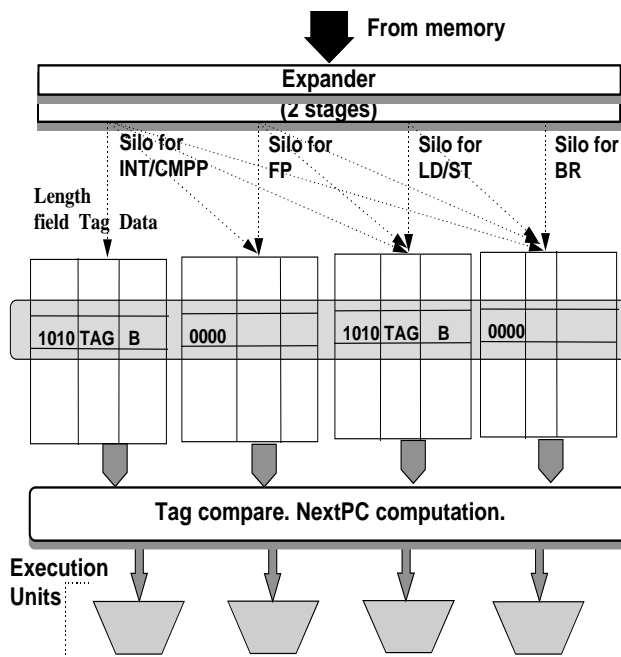
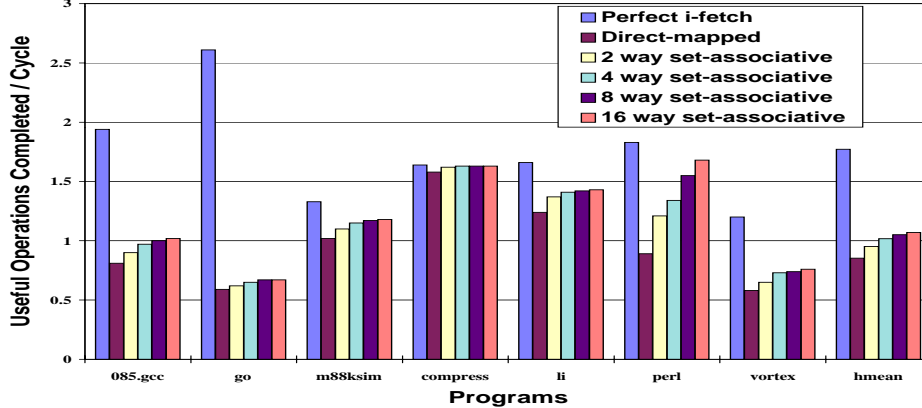


Figure 17: Instruction fetch for the rigid silo cache. Each silo is searched independently on each cycle. Each silo holds Ops destined for only one FUType. A two cycle expander between the cache and lower levels of memory routes Ops to their appropriate silo. The most significant bit of each length field is a valid bit.

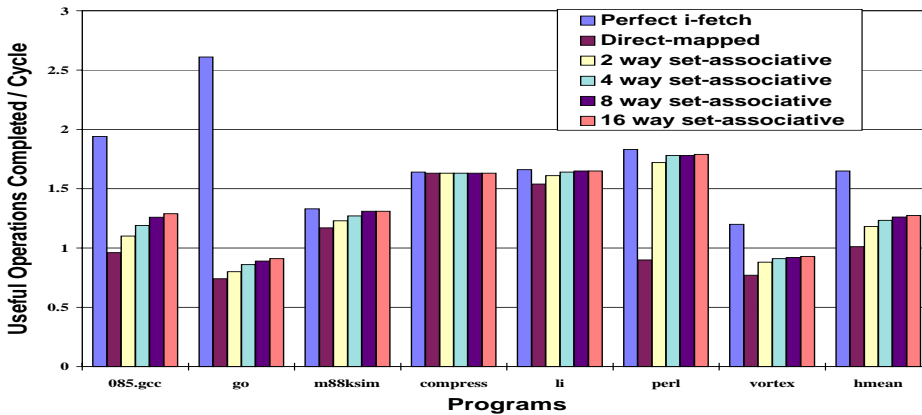
placements/replacements with a degree of autonomy. As shown in Figure 17, MultiOp *B* consists of an integer/compare-to-predicate Op and a memory Op. It does not place Ops into the silos for floating point or branch operations, and in fact those silos are empty as indicated by their valid bits. If a MultiOp *M*, containing only a floating point Op, maps to the same cache location, it is placed into the floating point silo and updates the appropriate fields for the proper location in that silo. MultiOps *B* and *M* can coexist at the same address across the silos.

The silo cache has a tag for every Op that is stored in the cache, in contrast to the uncompressed cache and the banked cache which use a tag for a cache block. The tag storage requirements for a silo cache are obviously higher. However, the data array for a silo cache need not hold all of the bits in an Op. Header and tail bits and the FUT field (see Figure 2) are no longer needed. For silo caches that hold the same number of Ops as 16KB and 32KB traditional caches, the overall storage requirements are approximately 63% greater.

Performance results for the rigid silo cache are shown in Figures 18 and 19. Across most benchmarks and cache sizes, the silo cache performs approximately the same as the banked cache, as indicated by the similarities in the harmonic means of their OPCs. In certain instances, there are noticeable differences: the banked cache performs better for 085.gcc for low associativities (direct-mapped through 4 way set-associative), but the silo cache performs as well or slightly better for 8 way and 16 way set-associative configurations. The only integer programs for which there are significant difference are *li* and *perl*. On *li*, the direct-mapped rigid silo cache outperforms the direct-mapped banked cache by 39% and 69% for 16KB and 32KB cache sizes, respectively. However, performance is nearly identical at higher associativities, and the banked cache actually performs better at higher associativities at the 16KB size - by 6% for 2 and 4 way associativity and by 7% for 8 and 16 way associativity. On *perl*, the rigid silo cache is a better performer



a) 16KB rigid silo cache, integer programs



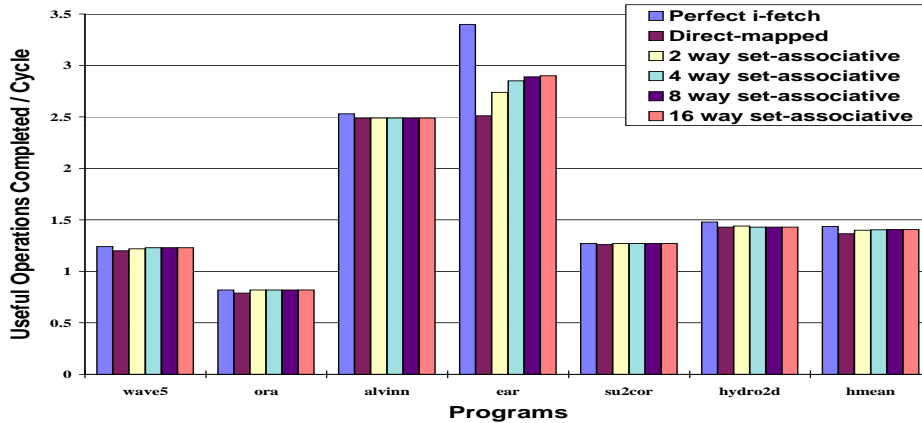
b) 32KB rigid silo cache, integer programs

Figure 18: Performance of the rigid silo cache on integer benchmarks. Figure a) shows the performance for a 16KB cache, and Figure b) shows the performance for a 32KB cache. The metric used is useful Ops completed per cycle (OPC). The set of bars on the far right of each graph is harmonic means.

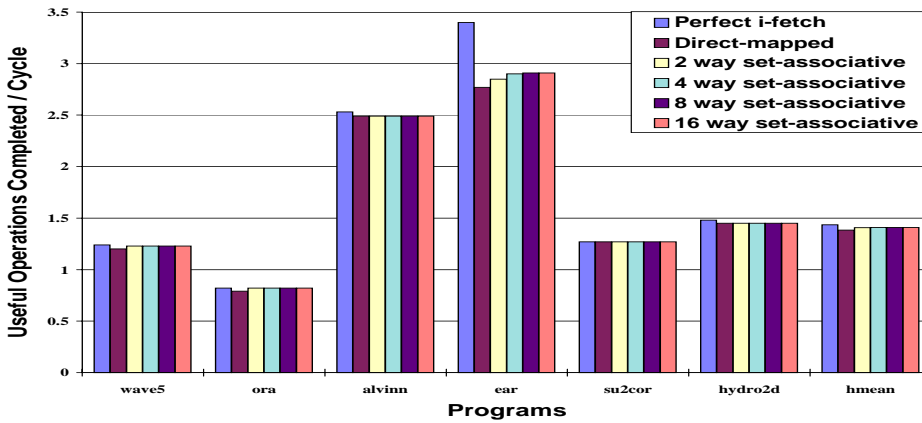
for 16KB caches across all associativities, by 16-45%. Also on perl, the rigid silo cache is a better performer for 32KB 2 way and 4 way set-associative configurations, by 44% and 25%, respectively. For the floating point programs, the performance of the rigid silo cache cache is again very similar to that of the banked cache. The only program for which there is a noticeable difference is ear. For a 16KB 2 way set-associative configuration, the rigid silo cache achieves an OPC 12% better than the banked cache.

4.3.2 The Flexible Silo Cache

The performance of the rigid silo cache is dependent on the FUType distribution in the workload. A program that does not contain any Ops mapping to a silo does not utilize that silo at all. In effect, the program sees a smaller i-cache, and the silos that are not used starve. For example, this behavior could occur in an integer-intensive application that starves the floating point silos. If the 1-FUType-to-1-silo requirement is relaxed so that complimentary types of Ops - those that typically do not execute together - are allowed to share a silo, the starvation problem might be eased. For example, if floating point and integer operations



a) 16KB rigid silo cache, floating point programs



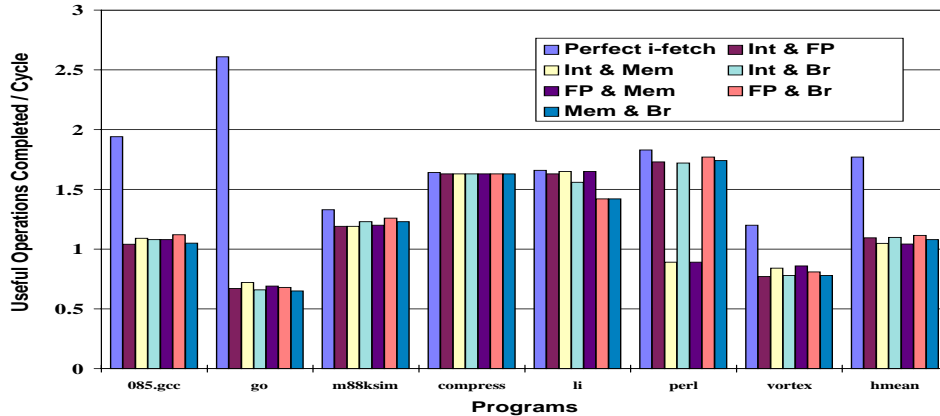
b) 32KB rigid silo cache, floating point programs

Figure 19: Performance of the rigid silo cache on floating point benchmarks. Figure a) shows the performance for a 16KB cache, and Figure b) shows the performance for a 32KB cache. The metric used is useful Ops completed per cycle (OPC). The set of bars on the far right of each graph is harmonic means.

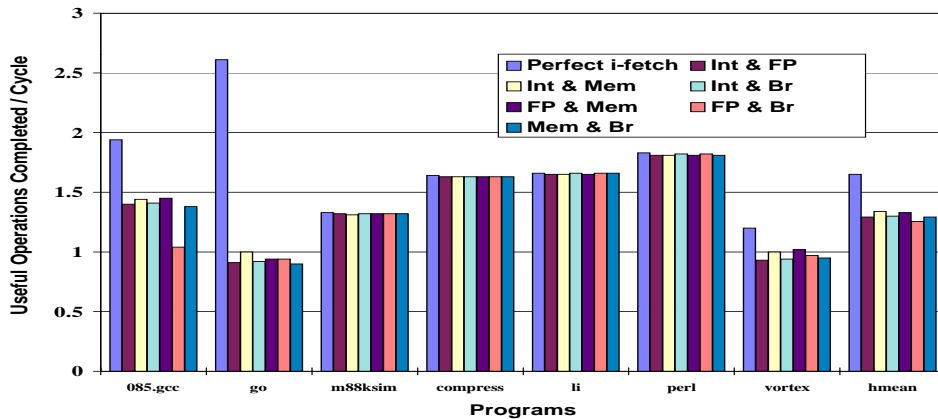
are placed in the same silo – an (I F) (M) (B) partitioning – the silo would be well utilized by programs that are floating-point or integer intensive as well as by programs that have a mix of both types of Ops. A small hit-path expander is required to route Ops from the silo to the appropriate functional units. This is in addition to the miss-path expander that is required for Op placement. The hit-path expander increases the branch misprediction penalty to two cycles.

A silo that allows multiple FUTypes per silo is a *flexible silo cache*, and a silo that holds multiple FUTypes is a *flexible silo*. The tag storage requirements for a flexible silo that supplies Ops for n FUs are the same as for the n silos for those FUs in a rigid silo cache. The data array is slightly larger because the FUT field is stored in the flexible silo, for use by the miss-path expander. A flexible silo holds Ops in a compressed fashion, but the overall design is nominally a NOP cache. The offset bits of Ops that map to a flexible silo are used to determine their locations within the silo.

A flexible silo design can have one or multiple flexible silos. To narrow the design space to a tractable size, this study considers cache configurations with one flexible silo that stores two FUTypes and the re-



a) 16KB 16 way set-associative flexible silo cache, integer programs

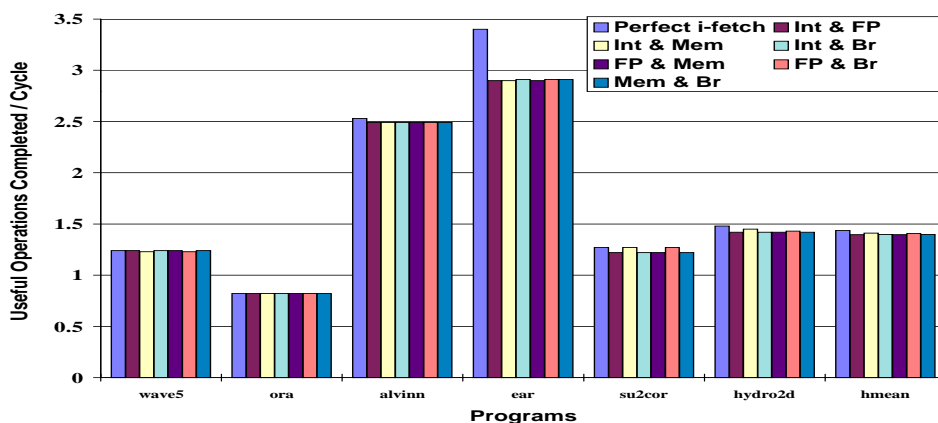


b) 32KB 16 way set-associative flexible silo cache, integer programs

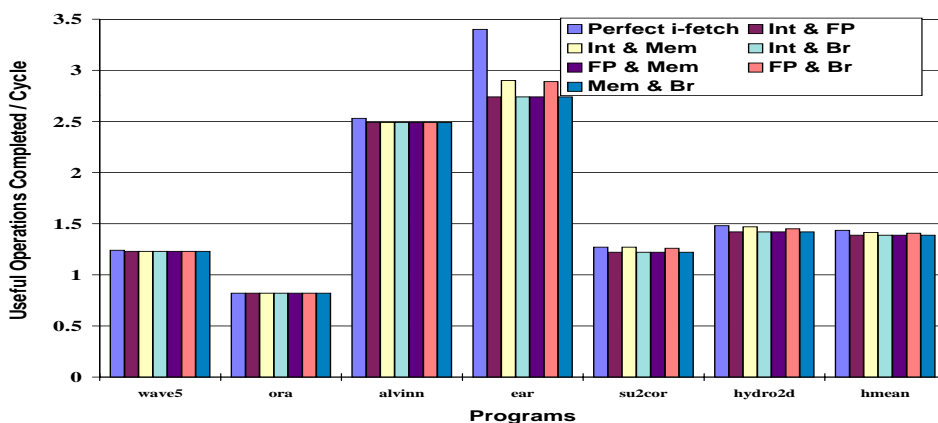
Figure 20: Performance of a 16 way set-associative flexible silo cache for integer benchmarks. Figure a) shows the performance for a 16KB cache, and Figure b) shows the performance for a 32KB cache. The metric used is useful Ops completed per cycle (OPC). The set of bars on the far right of each graph is harmonic means.

maining FUTypes are assigned to individual rigid silos. Simulations were performed with different flexible silo configurations to determine which FUTypes were best suited to share a flexible silo. Results for 16 way set-associative configurations are presented in Figures 20 and 21 (a complete set of results across all associativities is in the Appendix). The results can be compared with those for the rigid silo cache. Overall, the harmonic means for the rigid and flexible designs are approximately the same, across cache size and type of program. For the integer benchmarks, the largest difference in performance is for a 32KB cache on floating-point programs, where the flexible silo design outperforms the rigid silo design by about 5%. There are cases for individual programs where the differences are greater. For li and a 16KB cache, the Int-Mem, FP-Mem, Mem-Br, and Int-FP combinations perform better than the rigid design, by 9-15%. For perl and vortex and a 16KB cache, flexible configurations outperform the rigid design by 5-13%. Also on perl and a 16KB cache, however, the FP-Mem and Int-FP combinations perform 88% worse, respectively, than the rigid design. In certain cases, performance is very sensitive to the configuration of the flexible silo.

The results for different flexible combinations can be compared with each other. The harmonic means



a) 16KB 16 way set-associative flexible silo cache, floating point programs



b) 32KB 16 way set-associative flexible silo cache, floating point programs

Figure 21: Performance of a 16 way set-associative flexible silo cache for floating point benchmarks. Figure a) shows the performance for a 16KB cache, and Figure b) shows the performance for a 32KB cache. The metric used is useful Ops completed per cycle (OPC). The set of bars on the far right of each graph is harmonic means.

across both cache sizes and both types of programs show that the combinations differ slightly in their performance. The per-benchmark bars also show that performance did not vary greatly for most programs. There is a wide variance for perl, however. The FP-Mem and Int-FP combinations on the 16KB cache perform much worse than the other combinations, by up to 99%. This is due to the fact that perl has relatively few floating point Ops, and therefore a flexible silo with FP Ops will always perform poorly. However, this is contradicted by the good performance of the FP-Br flexible silo configuration. The conclusion is that perl has a high percentage of branch operations, so the FP-Br silo is heavily utilized by branch Ops and yields good performance. The only other notable case is 085.gcc on the 32KB cache, for which the Int-BR combination performs 32-39% worse than other combinations.

5 Analysis

In terms of complexity, the uncompressed cache is the simplest design. Its placement, addressing, and NextPC schemes are similar to techniques that have been used in previous designs. The drawback is that for the programs used for this study, the NOP policy causes low space utilization which results in poor performance. The banked cache uses a different approach. It can place multiple MultiOps in a cache block and does implicit prefetching. NextPC generation is performed differently than for the uncompressed cache but the hardware requirements of both mechanisms - an adder and extra storage for length/offset information - are equivalent. Overall, though, the banked cache requires more complex logic as it also must be able to interchange blocks from different banks and subsequently scan the Ops to later expand only the requested MultiOp. This extra work must be performed in one clock cycle and requires more (sequential) logic than the uncompressed design. The cycle time of the banked cache is potentially longer than that of the uncompressed cache. However, the performance is substantially better which indicates that the additional complexity may be worthwhile.

The rigid silo cache is nominally a NOP cache but allows limited sharing of cache locations by multiple MultiOps. It requires considerably more storage than either of the previous designs since each Op-wide storage location has associated with it a tag and a length field. Extra comparators are required to perform multiple parallel tag compares in each silo. The NextPC computation logic is identical to that used in the uncompressed cache. Overall, the rigid silo cache performs approximately the same as the banked cache but for some programs - `li` and `perl` - it performs better. From a purely cost/benefit perspective, the banked cache is a more efficient design. However, for applications that require high performance regardless of space constraints, the rigid silo cache is appropriate.

The flexible silo cache allows multiple FUTypes to reside in a silo. Conceptually, the design stores Ops in the flexible silo in a compressed fashion, so that the space utilization of the flexible silo is high. The extra expander required on the cache hit path is smaller than that used on the miss path and is a separate stage. Overall, the performance of the flexible design is equal to that of the rigid design. For the programs used in this study, performance across various combinations of FUTypes for the flexible silo varied very little. For `li`, `perl`, and `vortex`, certain flexible combinations outperformed the rigid silo cache, but other flexible combinations - FP-Mem and Int-FP for `perl` and the 16KB cache - performed worse. An interesting observation is that results for the flexible silo cache from this study differ from the results obtained in a previous initial study [13]. The previous results were obtained from a much smaller set of simulations, which explains the difference.

6 Related Work

This paper focuses on instruction fetch for compressed encodings, but other classes of encodings can also be used for VLIW architectures. The 4S architecture proposed by Sun Microsystems used a *frame encoding* which grouped MultiOps into instruction *frames*. An instruction frame is the same size as the i-fetch width of the machine. A frame encoding supports variable size instructions but enforces the restriction that all Ops in a MultiOp must reside in the same frame to ease the requirements on the i-fetch mechanism [14]. This requires NOPs, thereby violating RSI. The Cydrome Cydra 5 VLIW machine used a *split encoding* such that instruction cache blocks were composed of either one MultiOp or multiple, one-Op MultiOps called UniOps [15], [16]. Cache blocks composed of one MultiOp are in an uncompressed form and those composed of UniOps are padded with NOPs, if needed for cache block alignment. It is also non-RSI. Another

commercial VLIW architecture, the Multiflow TRACE family of machines, used a compressed encoding [17]. Nops were not stored in instruction words in memory. Instructions were expanded as they were fetched from memory into the instruction cache, as is done with a miss-path expander.

Much of the related work in instruction fetch mechanisms has concentrated on superscalar or CISC architectures, especially in the arena of x86 architectures. Patt, *et al.*, studied the use of the fill buffer and a decoded instruction cache to decompose CISC instructions into microoperations which can then be efficiently scheduled using dynamic scheduling hardware [18]. Smotherman and Franklin adapted the fill unit and decoded instruction cache for use in decoding x86 instructions [19]. Their design associates a NextPC field with each cache block. The Intel Pentium Pro processor employs a multi-stage i-fetch that fetches 16 bytes per cycle from the i-cache and uses three stages to align the instructions [5]. NextPC is PC+16 in the absence of a branch instruction [20]. The AMD K5 stores decode information related to instruction length in the L1 instruction cache which is later used for NextPC computation in the i-fetch stage [10]. Like the Pentium Pro, the K5 uses multiple stages to fetch and align an x86 instruction stream. An x86 processor design from NexGen uses a different approach for NextPC generation. It has dedicated logic that performs instruction alignment at fetch time to compute NextPC [21]. In the arena of RISC architectures, the CRISP processor used a decoded instruction cache [22]. CRISP instructions were converted from their in-memory format of 16-80 bits to a 192 bit expanded form in the i-cache. Each expanded instruction occupied a cache block by itself and had associated with it a NextPC field. The effect of encoding instructions in a compressed manner was studied by Wolfe and Chanin [23]. Their Compressed Code RISC Processor was designed to conserve memory bandwidth and did so by storing instructions in a compressed format in memory and decompressing them into the instruction cache at cache miss time. The R1 SPARC processor from HaL performs limited decoding between memory and the L1 i-cache to aid in decoding during i-fetch [24].

7 Conclusion

This study investigated the issues involved in i-fetch and i-cache support for VLIW architectures that use compressed encodings. The effect of i-fetch mechanisms on i-cache architecture was discussed, and a taxonomy for classifying VLIW i-caches based on degree of partitioning and the NOPs policy was introduced. Four cache designs were presented and evaluated: the uncompressed cache, the banked cache, the rigid silo cache, and the flexible silo cache. Performance and cycle times issues for each design was then discussed. The uncompressed cache was clearly the worst performer of the group, with OPCs up to 80% lower than the other designs. The banked cache and the rigid silo cache performed roughly equally, with the rigid silo cache yielding better OPCs on certain programs for certain cache configurations. The occasional gains in performance are at the cost of a much greater amount of storage hardware and tag comparison logic. The flexible silo cache performs almost equally to the rigid silo cache. Some combinations of the flexible silo perform slightly better (5-15%), but others perform much worse, on two occasions, 88% worse. The results indicate that the banked cache is the most effective design, from a cost/performance perspective. When implementation area is not an issue, a silo design can be used, and when application characteristics are well understood, a flexible silo design could be appropriate.

References

- [1] T. M. Conte and S. W. Sathaye, "Dynamic rescheduling: A technique for object code compatibility in VLIW architectures," in *Proc. 28th Ann. Int'l Symp. on Microarchitecture*, Ann Arbor, MI, Nov. 1995.

- [2] T. M. Conte et al., *The TINKER Machine Language Manual*, North Carolina State University, Raleigh, NC 27695-7911, Apr. 1995.
- [3] V. Kathail, M. Schlansker, and B. R. Rau, “HPL PlayDoh architecture specification: version 1.0,” Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, Technical Publications Department, 1501 Page Mill Road, Palo Alto, CA 94304, Feb. 1994.
- [4] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, “The Superblock: An effective structure for VLIW and superscalar compilation,” *The Journal of Supercomputing*, vol. 7, no. 1, pp. 229–248, Jan. 1993.
- [5] David B. Papworth, “Tuning the Pentium Pro microarchitecture,” *IEEE Micro*, vol. 16, no. 2, pp. 8–15, Apr. 1996.
- [6] William J. Bowhill et al., “A 300 mhz 64b quad-issue CMOS RISC processor,” In *Proc. 1995 Int’l Solid-State Circuits Conference* [25], pp. 182–183.
- [7] D. Alpert and D. Avnon, “Architecture of the Pentium microprocessor,” *IEEE Micro*, vol. 13, no. 3, pp. 11–21, June 1993.
- [8] Sanjeev Banerjia, Kishore N. Menezes, and Thomas M. Conte, “NextPC computation for a banked instruction cache for a VLIW architecture with a compressed encoding,” Technical report, Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27695-7911, June 1996, Available from <http://www.ece.ncsu.edu/tinker/nextpc.tr.ps>.
- [9] W. M. Johnson, *Super-scalar processor design*, Ph.D. thesis, Department of Electrical Engineering, Stanford University, Stanford, California, June 1989.
- [10] Dave Christie, “Developing the AMD-K5 architecture,” *IEEE Micro*, vol. 9, no. 2, pp. 16–26, 1996.
- [11] A. J. Smith, “Cache memories,” *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, 1982.
- [12] M. D. Hill and A. J. Smith, “Experimental evaluation of on-chip microprocessor cache memories,” in *Proc. 11th Ann. Int’l Symp. Computer Architecture*, Ann Arbor, MI, June 1984, pp. 158–166.
- [13] T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes, and S. W. Sathaye, “Instruction fetch mechanisms for VLIW architectures with compressed encodings,” in *Proc. 29th Ann. Int’l Symp. on Microarchitecture*, Paris, France, Dec. 1996.
- [14] S. Arya, H. Sachs, and S. Duvvuru, “An architecture for high instruction level parallelism,” in *Proc. 28th Hawaii Int’l. Conf. on System Sciences*, Maui, HI, Jan. 1995, pp. 153–161.
- [15] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, “The Cydra 5 departmental supercomputer,” *Computer*, vol. 22, no. 1, pp. 12–35, Jan. 1989.
- [16] G. R. Beck, D. W. L. Yen, and T. L. Anderson, “The Cydra 5 minisupercomputer: architecture and implementation,” *J. Supercomputing*, vol. 7, no. 1, pp. 143–180, Jan. 1993.

- [17] R. P. Colwell, R. P. Nix, J. J. O’Donnel, D. B. Papworth, and P. K. Rodman, “A VLIW architecture for a trace scheduling compiler,” in *Proc. Second Int’l. Conf. on Architectural Support for Programming Languages and Operating Systems.*, Palo Alto, CA, Oct. 1987, pp. 180–192.
- [18] S. Melvin, M. Shebanow, and Y. Patt, “Hardware support for large atomic units in dynamically scheduled machines,” in *Proc. 21th Ann. Int’l Symp. on Microarchitecture*, San Diego, CA, Dec. 1988, pp. 60–66.
- [19] Mark Smotherman and Manoj Franklin, “Improving CISC Instruction Decoding Performance Using a Fill Unit,” in *Proc. 28th Ann. Int’l Symp. on Microarchitecture*, Ann Arbor, MI, Dec. 1995, pp. 313–323.
- [20] David B. Papworth, ,” Personal communication, June 1996.
- [21] Donald Draper et al., “A 93 mhz, x86 microprocessor with on–chip L2 cache controller,” In *Proc. 1995 Int’l Solid-State Circuits Conference* [25], pp. 172–173.
- [22] D. R. Ditzel and H. R. McLellan, “Branch folding in the CRISP microprocessor: Reducing branch delay to zero,” in *Proc. 14th Ann. Int’l Symp. Computer Architecture*, Pittsburgh, PA, June 1987, pp. 2–9.
- [23] Andrew Wolfe and Alex Chanin, “Executing compressed programs on an embedded RISC architecture,” in *Proc. 25th Ann. Int’l Symp. on Microarchitecture*, Portland, OR, Dec. 1992, pp. 81–91.
- [24] Gene Shen et al., “A 64b 4–issue out–of–order execution RISC processor,” In *Proc. 1995 Int’l Solid-State Circuits Conference* [25], pp. 170–171.
- [25] *Proc. 1995 Int’l Solid-State Circuits Conference*, San Francisco, CA, Feb. 1995.

Complete results for the flexible silo cache

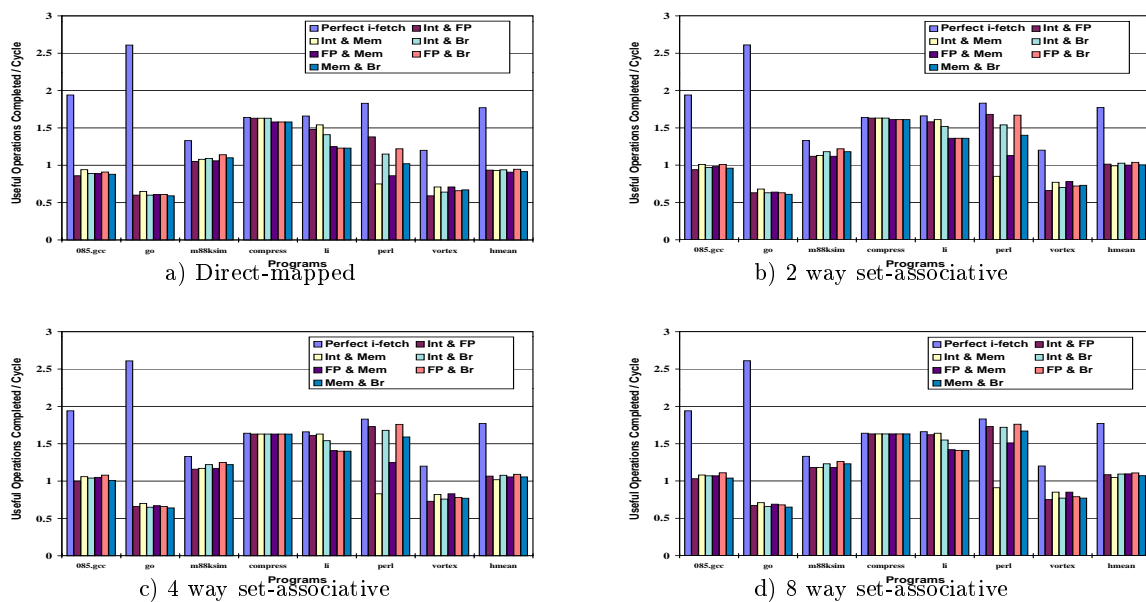
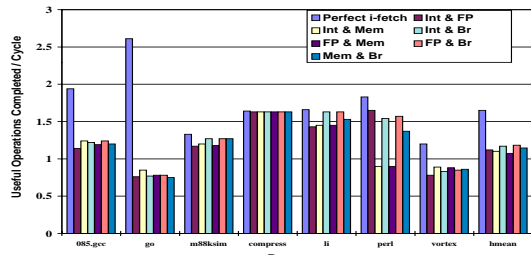
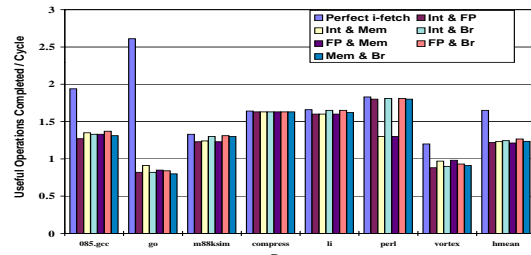


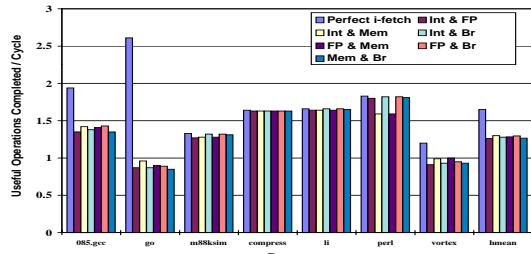
Figure 22: Performance of the 16KB flexible silo cache for integer benchmarks. Each graph shows performance for a particular set associativity. The metric used is useful Ops completed per cycle (OPC). The set of bars on the far right of each graph is harmonic means.



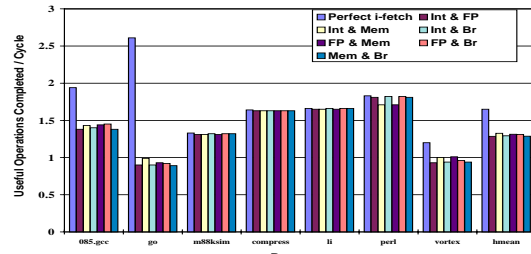
a) Direct-mapped



b) 2 way set-associative

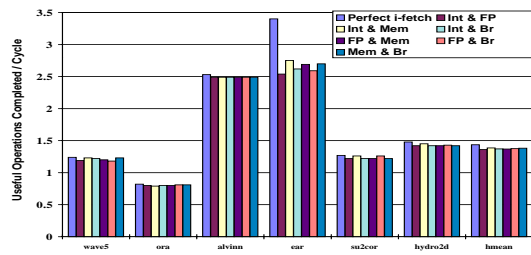


c) 4 way set-associative

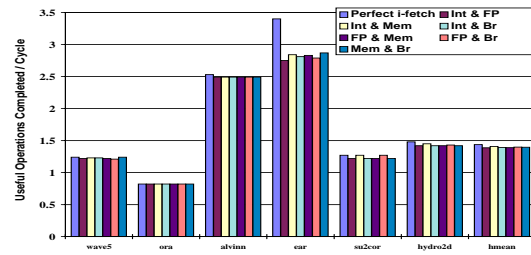


d) 8 way set-associative

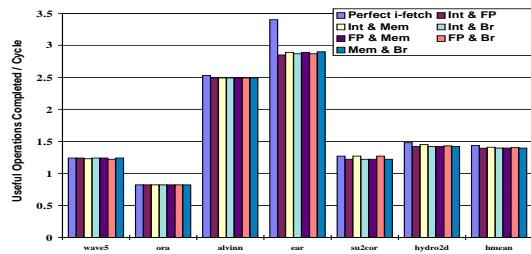
Figure 23: Performance of the 32KB flexible silo cache for integer benchmarks. Each graph shows performance for a particular set associativity. The metric used is useful Ops completed per cycle (OPC). The set of bars on the far right of each graph is harmonic means.



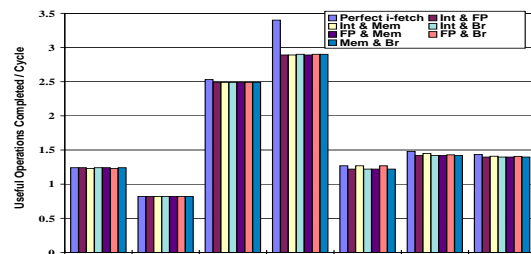
a) Direct-mapped



b) 2 way set-associative



c) 4 way set-associative



d) 8 way set-associative

Figure 24: Performance of the 16KB flexible silo cache for floating point benchmarks. Each graph shows performance for a particular set associativity. The metric used is useful Ops completed per cycle (OPC). The set of bars on the far right of each graph is harmonic means.

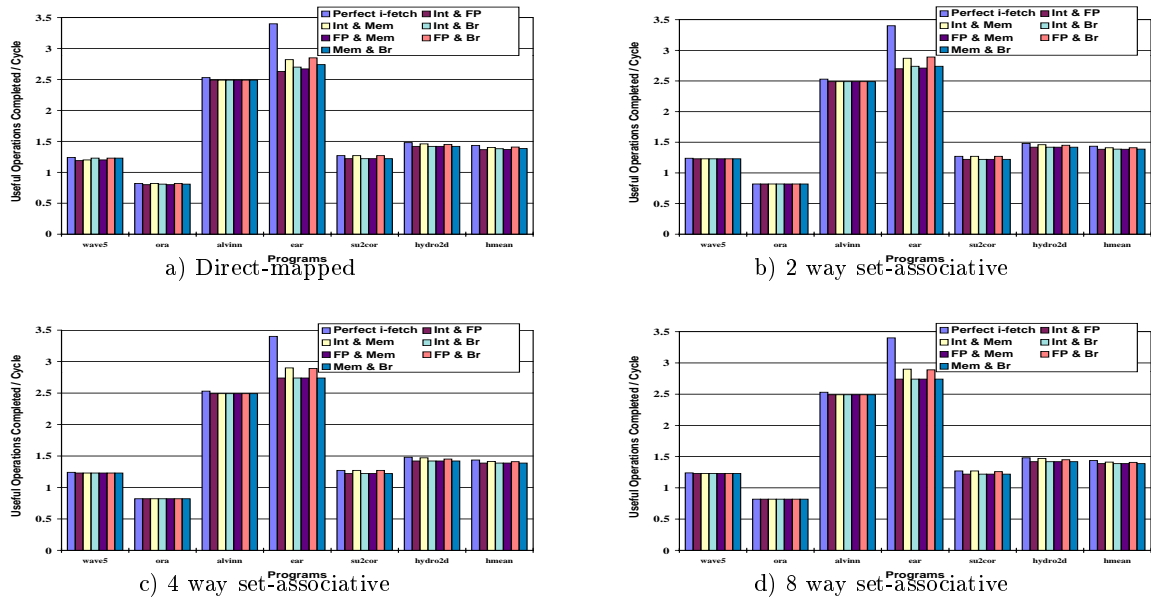


Figure 25: Performance of the 32KB flexible silo cache for floating point benchmarks. Each graph shows performance for a particular set associativity. The metric used is useful Ops completed per cycle (OPC). The set of bars on the far right of each graph is harmonic means.