Instruction Fetch Mechanisms for VLIW Architectures with Compressed Encodings *

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, North Carolina 27695-7911
(919)-515-5067
{conte,sbanerj,sylarin,knmeneze,swsathay}@eos.ncsu.edu

Abstract

VLIW architectures use very wide instruction words in conjunction with high bandwidth to the instruction cache to achieve multiple instruction issue. This report uses the TINKER experimental testbed to examine instruction fetch and instruction cache mechanisms for VLIWs. A compressed instruction encoding for VLIWs is defined, and a classification scheme for i-fetch hardware for such an encoding is introduced. Several interesting cache and i-fetch organizations are described and evaluated through trace-driven simulations. A new i-fetch mechanism using a silo cache is found to have the best performance.

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. The complexity of the hardware support required for i-fetch is related to the type of instruction encoding used. 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 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 [9]. A drawback is that such an encoding requires more complicated i-fetch 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 4.1-4.3 describe each of the i-cache designs in detail, including their respective performance. Section 6 discusses the performance of the designs and also discusses directions for future research.

2. A Basic Instruction Fetch Model

2.1. Instruction Fetch with a Compressed Encoding

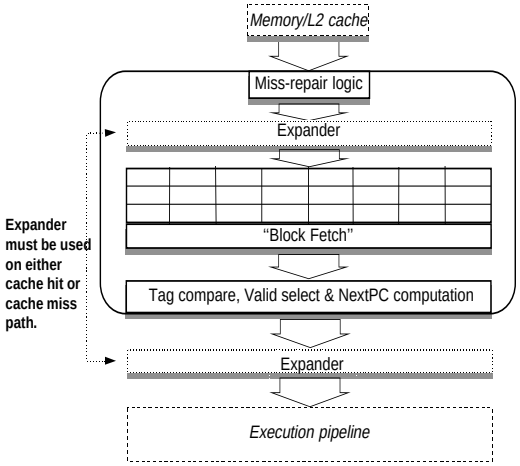

Figure 1. The basic instruction fetch model.
All of the necessary stages for i-fetch for a compressed encoding are shown.

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 is pipelined and consists of a *"block fetch"* stage that selects and fetches a cache block when presented with an address, and a *"tag compare & valid select"* stage that performs tag comparisons in parallel with selecting the Ops in the block that belong to the requested VLIW instruction.

A compressed instruction encoding using the TINKER VLIW experimental testbed is used for this study [8]. 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. A TINKER Op can execute

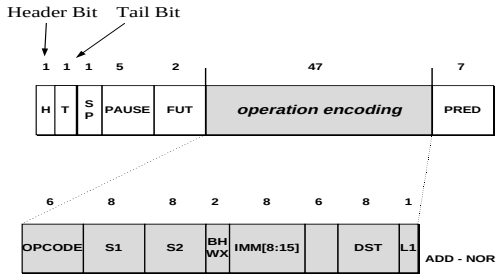

Figure 2. An integer add Op in the TINKER encoding.

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 Play-Doh specification from Hewlett Packard Laboratories [16]. 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.

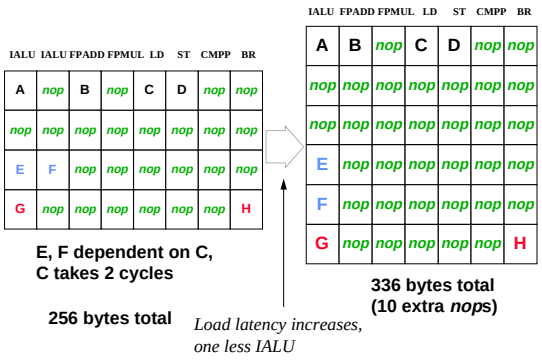

Figure 3. Object code compatibility using an uncompressed encoding.

The size of the code changes when it is rescheduled for a different machine organization.

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 [9]. 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 architecture[1].

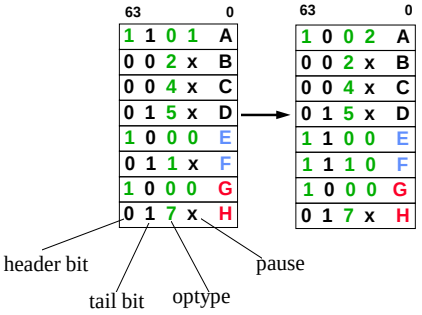

Figure 4. Object code compatibility using the TINKER compressed encoding.

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.

Although a compressed encoding has several advantages, it also requires a complex i-fetch mechanism. 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 for what FUType it is destined. An Op destined for a particular FUType 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. For example, if the instruction issue logic expects only integer Ops in Op position one in a MultiOp, and a one Op MultiOp is composed of a floating point Op, the Op must be routed to the floating point unit and not the integer 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. The number of extra stages is equal to the number of stages needed for expansion. 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. For this reason, hit path expansion should be performed in one cycle, requiring a complex and potentially costly implementation to meet the single cycle constraint.

2.2. Related Work

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 [3]. 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 [20], [5]. 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 (TRACE machines used cache miss expansion).

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

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 break down 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 [23]. 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 then uses three stages to align the instructions [19]. NextPC is PC+16 in the absence of a branch instruction. 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 [7]. 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 [11]. In the arena of RISC architectures, the CRISP processor used a decoded instruction cache [10]. 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 [24]. 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 [21].

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 memories). 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 searched[2]. 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 2.1, 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 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 i-cache pipeline (this is explained further in Section 4).

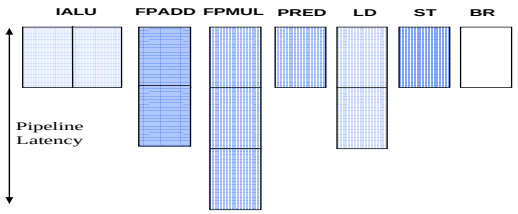

Figure 5. The TINKER-8 machine model.

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

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

[2]Set associativity permits parallel tag compares, but this is orthogonal to partitioning, as is shown in Section 4.3.

DoP = (I F M B), and uses a NOPs-free policy. The **silo cache** has a DoP = (I) (F) (M) (B) (fully partitioned), and a NOP policy. Lastly, the **flexible silo cache** uses a combined partition for two FUTypes and separate partitions for the remaining two FUTypes, and a NOP policy. The four designs are assumed to have the same cycle time. Trace driven simulations were used to evaluate the performance of the schemes. The IMPACT compiler employing superblock-based scheduling was used to compile benchmark programs using the TINKER compressed encoding for the TINKER-8 machine organization [13]. 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 [19], [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 suiteswere used as benchmarks for the evaluations and are listed in Table 1[3]. Two million Ops were sampled across the entire program for each simulation run.

4. Instruction Fetch Mechanisms

4.1. I-Fetch using the Uncompressed Cache

Figure 6 shows the organization of the *uncompressed cache*. The cache block size is the same as the machine width. A miss path expander is used with the cache miss-repair mechanism (a two cycle expander is assumed). The branch misprediction penalty is one cycle.

A modified direct mapped addressing scheme is used for the uncompressed cache. An example illustrates why traditional direct-mapped addressing (bit selection) cannot be used. Assume a TINKER-4 machine with a 32 KB i-cache that uses 32 bit addresses. Consider the two MultiOps in Figure 7, shown as they would appear in memory. All Ops in the first MultiOp are marked "A", and those in the second MultiOp are marked "B". MultiOp A has 2 Ops, while MultiOp B has a single Op. Assume that A does not contain a branch Op. If the cache were addressed as a normal direct-mapped cache, with the physical addresses of MultiOps A and B as 0x80000000 and 0x80000080, respectively, both would have the same tag and index, and hence map onto the same cache block (depicted in Figure 8(a)). For a sequential access pattern, a conflict miss is generated for the second MultiOp access i.e., for B. Not only do MultiOps A and B create contention for a cache block, but because their tags

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

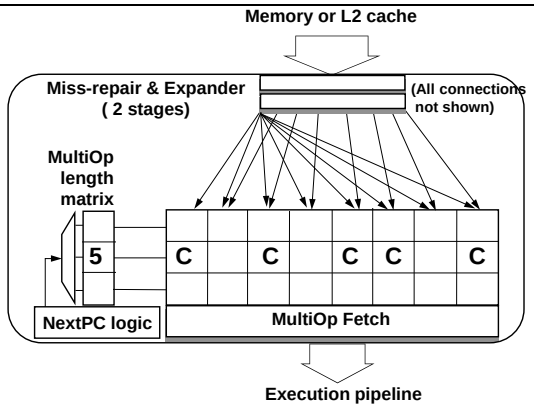

Figure 6. The uncompressed cache.
A two cycle miss path expander is used to place Ops into positions within a cache block. A length field is associated with every cache block.

are identical it cannot be determined which one is resident in the cache.

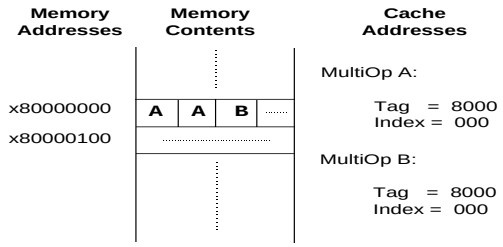

Figure 7. A traditional address mapping for the uncompressed cache.

One way to differentiate between such sequential Ops is to use some of the offset bits as part of the index. Figure 8(b) depicts how the cache interprets the address when using such a scheme. This *offset reduced* addressing maps each individual MultiOp from a memory block frame into a separate cache block. Associativity is not required because the possibility of sequential MultiOps mapping to the same cache block is eliminated.

The uncompressed cache holds the MultiOp in uncompressed form, hence the valid select logic and the hit path expander that were shown as a stage in the basic model (Figure 1) are unnecessary. NextPC computation still needs to be performed. The fetch hardware must add the length of the current MultiOp to the PC to generate the NextPC. Computation of the length can be performed at run-time, after the block containing the MultiOp is fetched out of the cache matrix. But this places the NextPC computation *after* the block fetch, adding sequential computation at the end of fetch cycle and possibly increasing the cycle time. To avoid

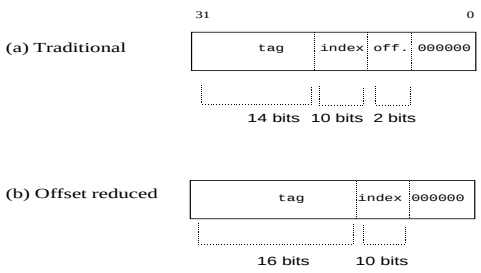

Figure 8. Interpreting a cache address – (a) traditional scheme, (b) reduced offset scheme.

this situation, a MultiOp length field can be associated with every cache block. The length field can be accessed in parallel with the tag and data, and the addition of the length to the PC can be done before the beginning of the tag compare stage.

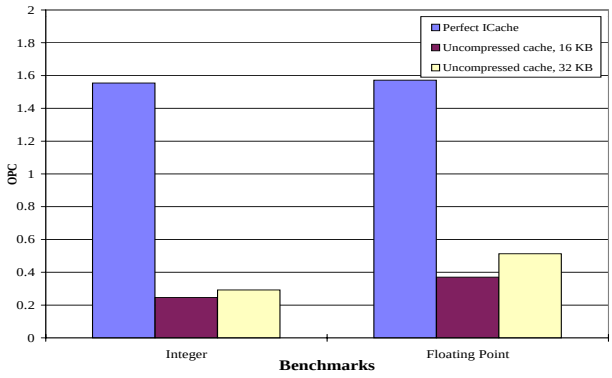

Figure 9. Performance of the uncompressed cache.

The metric used in useful Ops completed per cycle (OPC). Each individual bar represents the harmonic mean for the combination of cache size and benchmark programs.

The graph in Figure 9 presents the results of simulations for the performance of the uncompressed cache as compared to a perfect i-cache (a cache that never misses). The metric used here – and for all simulations in this report – is useful Ops completed per cycle (OPC). OPC is used because it captures the effect of i-cache performance on overall performance. The results are grouped based on cache size and the type of benchmarks. The uncompressed cache performs very 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 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 slower than a perfect i-cache. For the floating point benchmarks, the larger cache performed 19% better but is still four times slower than the perfect i-cache. The reason for this 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. A possible solution to this is a design that can place more than one MultiOp in a cache block, and this is explored in the next section.

4.2. I-Fetch using the Banked Cache

Figure 10 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 [2]. The cache block size is the same as the machine width n. 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 adds an extra stage to the processor pipeline. The extra stage increases the branch misprediction penalty to two cycles.

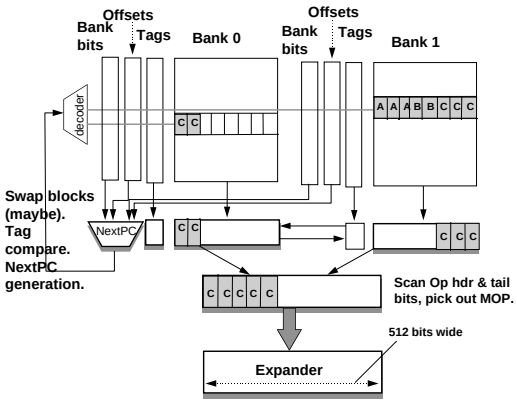

Figure 10. The banked cache.

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 memory, 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 address the cache, the cache address decoder selects consecutive blocks in both cache banks.

The fetch mechanism is shown in Figure 10 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 fetch cycle. There are three sequential steps required to fetch MultiOp C. These are also shown in Figure 10 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 be accomplished by using extra bits to store offset information for MultiOps in the cache. Details of the hardware are pictured in Figure 11. An offset field and a bank bit are maintained for every Op in a cache block. The offset field indicates the offset within the cache block of the next MultiOp, and the bank bit indicates if the next MultiOp resides in the same bank as the current MultiOp or in the next bank. The values for these fields are set as Ops are received in the cache–memory interface at cache miss time. (The complete algorithm for NextPC computation is available in a technical report [4].) Offset fields and bank bits can also be used in a two-way set associative design, in tandem with way prediction techniques [15]. 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 [14] and implemented in the AMD K5 [7], 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 a 32 KB direct-mapped configuration, the total size of the cache (tag and data arrays) is increased by approximately 8% over that of a traditional cache.

Figure 12 presents the results for simulations of the banked cache compared against a perfect i-cache. The banked design performs at least an order of magnitude better than the uncompressed design. The greater utilization

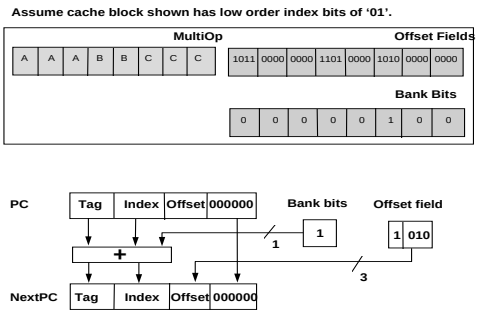

Figure 11. NextPC computation for the banked cache.

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.

of the data storage area by the banked design offsets the greater branch misprediction penalty. 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* [22]. Sub-blocking partitions each cache block into smaller units (sub-blocks). Sub-blocking usually increases the miss ratio of a cache but reduces the amount of memory traffic as each sub-block in a 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.

Prefetching can be used in a sub-blocked banked cache. Block fetching performs *implicit prefetching*: as the requested MultiOp is loaded into the cache, fragments of other

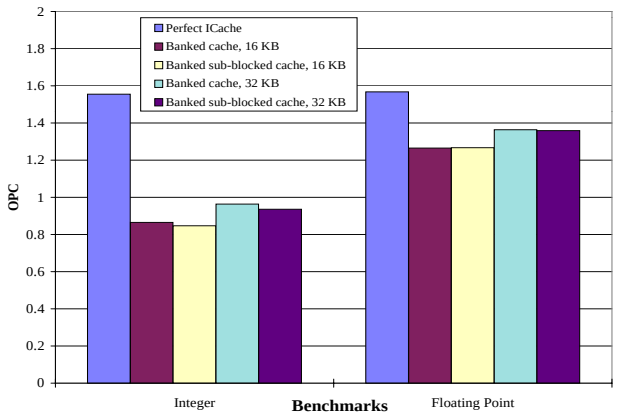

Figure 12. Performance of the banked cache.
The metric used is useful Ops completed per cycle (OPC). Each individual bar represents the harmonic mean for the combination of cache size and benchmarks programs.

MultiOps that reside in the same block(s) are also loaded. 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 X lies and all subsequent sub-blocks to the end of the cache block.

A banked i-cache with a one Op sub-block size was simulated and the results are presented in Figure 12. The performance of the sub-blocked design was very benchmark dependent. Sub-blocking was very effective for the integer programs, outperforming the non-sub-blocked design by 9% and 45% for 16 KB and 32 KB sizes, respectively. For the floating point benchmarks, the performance was slightly lower than that of the non-sub-blocked design. Examining the behavior of the programs revealed the reasons for the differences in performance. The integer programs have many branches that branch to the latter half of a cache block. When such a branch causes a cache miss, only the latter half of the block and the next block are filled. Contrast this with a block fetch design in which both blocks in their entirety are fetched. The fill time for the sub-blocked design is potentially up to almost 50% less. For the floating point programs, the total number of stall cycles due to i-cache misses were slightly higher. 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 beginning words 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.

4.3. I-Fetch using 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 13. 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 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 the sets.

4.3.1. The Rigid Silo Cache

The *rigid silo cache* is pictured in Figure 13. 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.

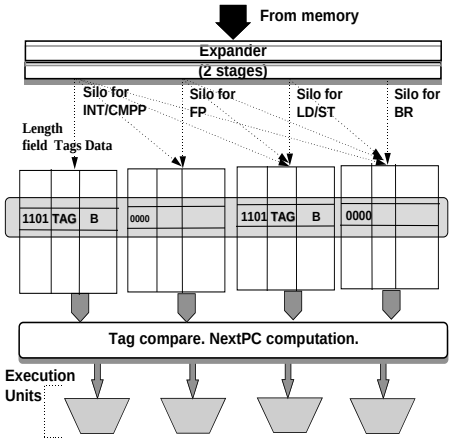

Figure 13. Instruction fetch for the rigid silo cache.

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.

As with the uncompressed cache, in a silo cache sequential MultiOps of a size less than the machine width can map into 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, similar to the banked cache (as shown for MultiOp B in Figure 13, the tags and length bits for all Ops in 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 miss occurs when either none of the tags in any of silos match or a tag match occurs but the length 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 and length fields 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 need to be flagged to indicate that the entire "parent" MultiOp is not present. This is accomplished by searching for all Ops that have the same tags as the Ops being replaced, and un-setting the length valid bits for those Ops. The next cache access for the partially displaced MultiOp results in a miss, due to the un-set length 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 perform placements/replacements with some degree of autonomy. As shown in Figure 13, MultiOp B consists of a 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 these silos are currently empty as indicated by their valid bits. If a MultiOp X, containing only a single floating point operation, maps to the same cache location, it can be placed into the floating point silo and can update the tag and length fields for the proper location in that silo. MultiOps B and X 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, all of 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 a Op. Header and tail bits and the FUT field (see Figure 2) are no longer needed. For a TINKER-8 machine, a 32 KB silo cache requires approximately 10 KB (or 30%) more storage than a traditional cache.

Performance results for the rigid silo cache are shown in Figure 14. Even at low degrees of associativity (direct-mapped and two-way), the silo cache outperforms the banked cache by 6-18%. For the integer programs, higher degrees of associativity reduces the number of conflict misses and yields further improvements.

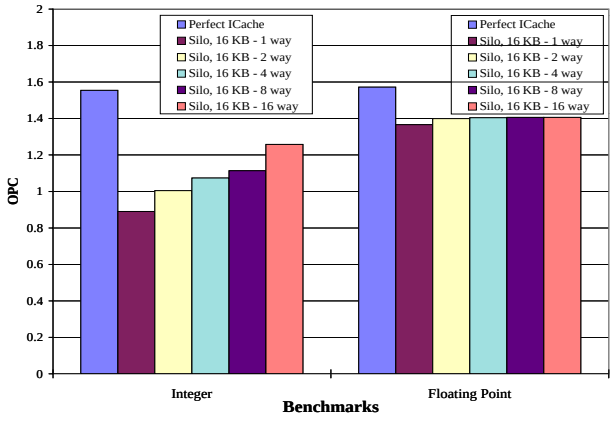

Figure 14. Performance of the rigid silo cache.

The metric used is useful Ops completed per cycle (OPC). Each individual bar represents the harmonic mean for the combination of cache size and benchmarks programs.

4.3.2. The Flexible Silo Cache

The performance of a silo cache is dependent on the FU-Type 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 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 termed a *flexible silo cache*, and a silo that holds multiple FUTypes is a *flexible silo*. A flexible silo that supplies Ops for n FUs has a tag array of the same size as 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, but not in the rigid design. A flexible silo holds Ops in a compressed fashion, but the overall design

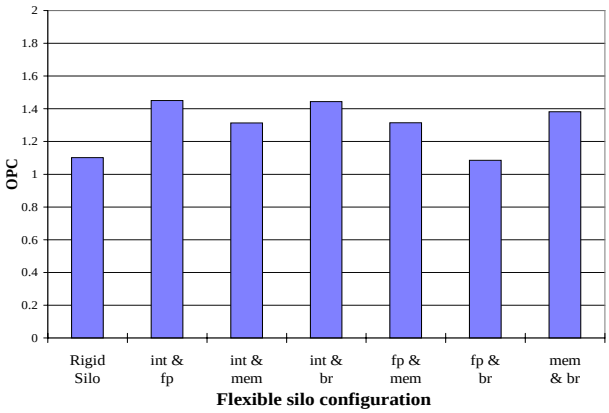

Figure 15. Evaluation of shared silos for integer programs.

The metric used is useful Ops completed per cycle (OPC). With the exception of the first bar on the chart, each individual bar represents the harmonic mean for the combination of benchmarks programs using one flexible silo for the FUTypes indicated and private silos for the remaining FUTypes. The caches simulated were 16 KB with associativities of 1, 2, 4, 8, and 16. The first bar represents the harmonic mean for all benchmarks using the same cache size and associativity parameters but with a rigid silo design.

is nominally a NOP cache. The offset bits of Ops that map to a flexible silo are used to determine their cache locations. A flexible silo that holds m FUTypes has the same size tag and data arrays as the m silos for those FUTypes in a rigid silo cache. Of particular interest is a design that stores two FUTypes in a flexible silo and assigns the remaining two FUTypes to 1-FUType-per-partition silos.

Simulations were performed for the integer benchmark programs for a 16 KB cache with a variety of flexible partitions to determine which FUTypes were best suited to share a flexible silo. The results are presented in Figure 15. With the exception of the floating point/branch combination, all of the shared silo designs performed better than the rigid silo design. The performance gains ranged from 19% for the integer/memory combination to 31% for the integer/floating point combination. Based on these preliminary results, the flexible silo design clearly outperforms the rigid silo design as well as the uncompressed and banked cache designs.

5. Analysis

In terms of complexity, the uncompressed cache can be viewed as the simplest i-fetch mechanism. It's Op placement, Op access, and NextPC schemes are similar to techniques that have been used in previous designs. The drawback of this simplicity is that, for the workload presented in this study, the NOP policy causes low space utilization which is reflected by 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 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 extra complexity may be warranted.

The rigid silo cache is nominally a NOP cache but allows limited sharing of cache locations by multiple MultiOps. The rigid silo cache requires considerably more storage than either of the previous designs as each Op-wide storage location has associated with it a tag and a length field. Extra comparators are also required to perform multiple parallel tag compares in each silo. The NextPC computation logic is identical to that used in the uncompressed cache. A direct–mapped silo cache performs roughly equivalent to a banked cache. As associativity is increased, the rigid silo cache outperforms the banked cache. However, set associativity requires extra levels of logic to select the correct block from within the set.

The flexible silo design allows multiple FUTypes to reside in a silo. Conceptually, the design stores Ops in the flexible silo in a compressed fashion, so that space utilization of the flexible silo is high. Cache placement and access are straightforward and is no more complex than that for the other designs. The extra expander required on the cache hit path is smaller than that used on the miss path (and the hit-path expander used in the banked cache) and is a separate stage. If set associativity is used, the cycle time might be stretched due to the extra levels of logic required. The preliminary evaluation of the flexible silo cache indicates that it might outperform the other three designs. Further work is needed to determine if the (potentially) greater performance is not countered by an increase in cycle time.

6. Conclusion

This study investigated the issues involved in i-fetch 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. The performance and cycle times issues for each design was then discussed. The silo-based designs proved to be the best performers, with the flexible silo design showing promise in a preliminary evaluation. Based on the results, the plan for the TINKER testbed is to complete the evaluation of the flexible silo design and investigate the implementation details and tradeoffs for the rigid silo and flexible silo cache designs.

Acknowledgments

Discussions with the other members of the TINKER group proved useful to the development of ideas in this paper. This work was supported by the Intel corporation and the National Science Foundation under grants MIP-9696010 and MIP-9625007. The authors would like to express their gratitude to the University of Illinois IMPACT group for the use of the IMPACT compiler system. The comments from the anonymous referees are also appreciated.

References

[1] *Proc. 1995 International Solid-State Circuits Conference*, San Francisco, CA, Feb. 1995.

[2] D. Alpert and D. Avnon. Architecture of the Pentium microprocessor. *IEEE Micro*, 13(3):11–21, June 1993.

[3] S. Arya, H. Sachs, and S. Duvvuru. An architecture for high instruction level parallelism. In *Proc. 28th Hawaii Int'l. Conf. on System Sciences*, pages 153–161, Maui, HI, Jan. 1995.

[4] S. Banerjia, K. N. Menezes, and T. M. Conte. NextPC computation for a banked instruction cache. Technical report, Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27695-7911, June 1996.

[5] G. R. Beck, D. W. L. Yen, and T. L. Anderson. The Cydra 5 minisupercomputer: architecture and implementation. *J. Supercomputing*, 7(1):143–180, Jan. 1993.

[6] W. J. Bowhill et al. A 300 mhz 64b quad-issue CMOS RISC processor. In *Proc. 1995 International Solid-State Circuits Conference* [1], pages 182–183.

[7] D. Christie. Developing the AMD–K5 architecture. *IEEE Micro*, 9(2):16–26, 1996.

[8] T. M. Conte et al. *The TINKER Machine Language Manual*. North Carolina State University, Raleigh, NC 27695-7911, Apr. 1995.

[9] T. M. Conte and S. W. Sathaye. Dynamic rescheduling: A technique for object code compatibility in VLIW architectures. In *Proc. 28th Ann. International Symposium on Microarchitecture*, Ann Arbor, MI, Nov. 1995.

[10] D. R. Ditzel and H. R. McLellan. Branch folding in the CRISP microprocessor: Reducing branch delay to zero. In *Proc. 14th Ann. International Symposium Computer Architecture*, pages 2–9, Pittsburgh, PA, June 1987.

[11] D. Draper et al. A 93 mhz, x86 microprocessor with on–chip L2 cache controller. In *Proc. 1995 International Solid-State Circuits Conference* [1], pages 172–173.

[12] M. D. Hill and A. J. Smith. Experimental evaluation of on–chip microprocessor cache memories. In *Proc. 11th Ann. International Symposium Computer Architecture*, pages 158–166, Ann Arbor, MI, June 1984.

[13] 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*, 7(1):229–248, Jan. 1993.

[14] W. M. Johnson. *Super-scalar processor design*. PhD thesis, Department of Electrical Engineering, Stanford University, Stanford, California, June 1989.

[15] T. Juan, T. Lang, and J. J. Navarro. The difference–bit cache. In *Proc. 23nd Ann. International Symposium Computer Architecture*, pages 114–120, Philadelphia, PA, May 1996.

[16] V. Kathail, M. Schlansker, and B. R. Rau. HPL PlayDoh architecture specification: version 1.0. Technical Report HPL-93-80, Hewlett-Packard Laboratories, Technical Publications Department, 1501 Page Mill Road, Palo Alto, CA 94304, Feb. 1994.

[17] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donell, and J. C. Ruttenberg. The Multiflow Trace scheduling compiler. *J. Supercomputing*, 7(1):51–142, Jan. 1993.

[18] S. Melvin, M. Shebanow, and Y. Patt. Hardware support for large atomic units in dynamically scheduled machines. In *Proc. 21th Ann. International Symposium on Microarchitecture*, pages 60–66, San Diego, CA, Dec. 1988.

[19] D. B. Papworth. Tuning the Pentium Pro microarchitecture. *IEEE Micro*, 16(2):8–15, Apr. 1996.

[20] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The Cydra 5 departmental supercomputer. *Computer*, 22(1):12–35, Jan. 1989.

[21] G. Shen et al. A 64b 4–issue out–of–order execution RISC processor. In *Proc. 1995 International Solid-State Circuits Conference* [1], pages 170–171.

[22] A. J. Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.

[23] M. Smotherman and M. Franklin. Improving CISC Instruction Decoding Performance Using a Fill Unit. In *Proc. 28th Ann. International Symposium on Microarchitecture*, pages 313–323, Ann Arbor, MI, Dec. 1995.

[24] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *Proc. 25th Ann. International Symposium on Microarchitecture*, pages 81–91, Portland, OR, Dec. 1992.