NextPC computation for a banked instruction cache for a VLIW architecture with a compressed encoding

Sanjeev Banerjia Kishore N. Menezes Thomas M. Conte

Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, North Carolina 27695-7911
(919)-515-7983
E-mail: {sbanerj,knmeneze,conte}@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. One instruction fetch mechanism for VLIWs is the use of a *banked instruction cache*. Such a cache is intended for use with a compressed instruction encoding. A compressed encoding supports variable length VLIWs and thus has associated with it the difficulties in supporting variable length instructions. One of these is determining on every cycle the instruction fetch address (NextPC) for the following cycle. This report uses the TINKER experimental testbed to illustrate a mechanism that can be used by a banked instruction cache for NextPC computation. An algorithm for NextPC computation is outlined and associated hardware support is presented. Issues relating to the cycle time complexity of the proposed design are also addressed.

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 a wide instruction word on every clock cycle to function units (FUs) in the machine. Two classes of encodings for VLIWs are *uncompressed* and *compressed* encodings. In an uncompressed encoding, NOP operations are explicitly stored 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 in that instruction word. An uncompressed encoding implies that all instructions are of a fixed length, which can simplify the instruction fetch hardware but at the expense of poor memory utilization.

On the other hand, compressed encodings do not store NOPs. Only operations for the FUs that will execute an operation are included in the instruction word. NOPs are effectively compressed out of the instruction word. VLIWs encoded with a compressed encoding are variably sized. 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 instruction fetch to handle the variable length instructions.

Research has been performed to investigate the instruction fetch requirements of VLIWs that use compressed encodings [2]. One of the i-fetch mechanisms that has been proposed is a banked instruction cache. As alluded to previously, a compressed encoding requires hardware support to fetch the variably-sized instructions, support that is not needed to fetch the the fixed size instructions of a VLIW that uses an uncompressed encoding. One of the issues is determining the fetch address for the next clock cycle while fetching a (variably sized) instruction in the current cycle. This can be referred to as NextPC computation. This report outlines an algorithm for NextPC computation for a banked instruction cache. Hardware support for the algorithm is presented and im-

plementation issues are also addressed.

2 Instruction Fetch Mechanisms and Related work

2.1 Instruction Fetch Mechanisms

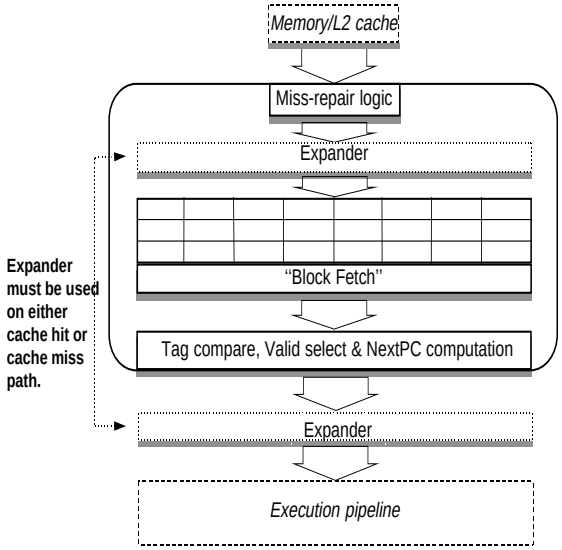


Figure 1: The basic instruction fetch model. All of the necessary stages for i-fetch for a compressed encoding are shown.

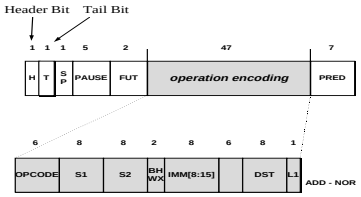


Figure 2: An integer add Op in the TINKER encoding.

A simplified instruction fetch model is shown in Figure 1. The solid lower borders in the diagram indicate pipeline latches that delineate the various stages of the fetch pipeline. The main blocks in this model are the miss-repair logic, the pipelined instruction cache, and the expander. 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 (for the purposes of this report, assume that each block can hold multiple VLIWs). 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. The expander stage is responsible for routing Ops to the FUs on which they are to be executed. Routing is necessary because the placement of Ops within a MultiOp is somewhat arbitrary when using a compressed encoding (for details, refer to Conte *et al.* [2]).

The TINKER compressed instruction encoding is used for this study [3]. TINKER is a VLIW encoding that combines individual operations (Ops) that can be issued in parallel into a unit of parallel issue called a MultiOp. A TINKER Op is a PlayDoh instruction that is 64 bits in length [4]. A TINKER Op can execute on one of four types of functional units (*FU-Type*): integer (integer computation and predicate handling), memory (loads and stores), FP (floating point add/mul/div/convert) and branch. 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. 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.

2.2 Related Work

Only two classes of encodings – uncompressed and compressed – have been introduced so far. Other classes of encodings can also be used for VLIW architectures, such as a frame encoding [5], split encoding [6], [7], and a packet encoding [8], among others. The more germane issue for this report is work related to instruction fetch mechanisms, particularly those that deal with variably sized instructions and NextPC computation. A review of the issue might help to illustrate the (potential) problem. One step of i-fetch is NextPC generation, during which a PC is generated for the subsequent i-cache access. NextPC generation for an encoding that supports fixed size instructions (such as 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 (such as an architecture that uses a compressed encoding) have to first determine the length of the current instruction being fetched to determine what quantity to add to

the PC to get NextPC.

Much of the related work in instruction fetch mechanisms has concentrated on superscalar or CISC architectures, especially in the arena of x86 architectures. Smotherman and Franklin adapted the fill unit and decoded instruction cache originally proposed by Patt *et al.* for use in decoding x86 instructions [9], [10]. Their design associates a NextPC field with each cache block, so that the NextPC is immediately available and does not need to be computed with every cache access. High performance CISC processors implement different solutions for fetching variable length instructions. 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 [11]. 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 [12]. Like the Pentium Pro, the K5 uses multiple stages to fetch and align an x86 instruction stream. The Nx86 processor design from NextGen uses a different approach for NextPC generation. It has dedicated logic that performs instruction alignment at fetch time to compute NextPC [13]. In the arena of RISC architectures, the CRISP processor used a decoded instruction cache to assist in the decoding of instructions [14]. CRISP instructions were converted from their in–memory format of 16 to 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.

3 An overview of the Banked Cache

Figure 3 shows the organization of the instruction fetch mechanism when using the banked cache. The cache is organized as two data and tag arrays, as in the Intel Pentium processor [15]. The cache block size is the same as the machine width n and the maximum size of a MultiOp is n Ops. MultiOps are stored in a compressed fashion in the cache. 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 (*successor block*). MultiOp expansion is done on every cache access by an expander stage that is located between the cache and the execution pipeline (this is called a *hit-path expander* as it used on every cache access).

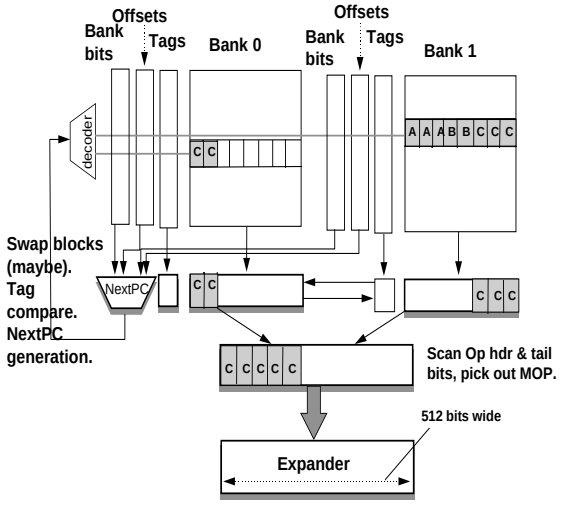


Figure 3: Instruction fetch for a MultiOp in the banked cache design. A fetch for MultiOp C is shown. Blocks from both banks are fetched and then swapped before being passed to the single-cycle expander. NextPC computation is done using offset and bank bits and is performed in parallel with cache access.

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 3 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 3 and detailed below.

1. The current block containing the first three Ops (in bank one) and the successor block containing the fourth and fifth Ops (in bank zero) 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, combined with the knowledge of the starting position, 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 [16].

3. The header bits for all Ops in the blocks are scanned to determine the Ops belonging to the MultiOp, 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 [16].

While fetching the current MultiOp, NextPC must also be computed for the instruction fetch on the next clock cycle. In the schemes described by Conte *et al.* [16], this was done via a BTB but none exists in TINKER. The next section describes NextPC computation for the banked cache.

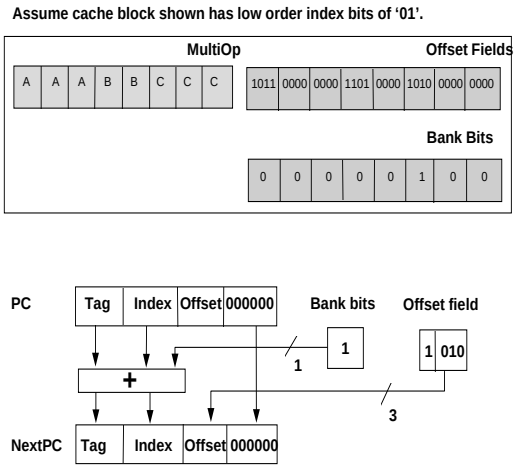


Figure 4: Hardware support for NextPC computation: A cache access for MultiOp C in a TINKER-8 banked cache is shown. A valid bit is associated with every offset field. Because C's offset field is valid, it is placed into the appropriate position for the NextPC address, and C's bank bit is added to the remaining high order bits to form a new index and tag.

4 NextPC computation

While fetching the current MultiOp, NextPC must also be computed. In the absence of an Op that changes the control flow of the program (such as a branch or a function call), this can be accomplished by using extra bits to store the offset of the next sequential MultiOps in the cache. The hardware support for NextPC computation is described and the algorithm is then introduced.

4.1 Hardware Support

Figure 4 is a conceptual depiction of the hardware required for NextPC computation (a more concrete discussion on implementation follows in Section 6). 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* sequential MultiOp, and is only set for the first Op in a MultiOp. The bank bit indicates if the next sequential MultiOp begins in the current bank or in the next bank: the bit is set to 1 if the next MultiOp begins in next bank, and to 0 if the next MultiOp begins in the same bank. The offset field is substituted into the appropriate bit positions of the current PC and the bank bit is added to the tag and index bit concatenation to form NextPC. For the TINKER encoding, the offset field is placed into positions PC_8-PC_6. An adder is required to add the bank bit to the tag and index bits of the PC to create the tag and index of NextPC.

For a banked cache that has a block size of n words (Ops), a $\log_2 n$ bit offset field per word is needed, plus a valid bit and a bank bit. Compared to a cache design without offset fields and bank bits, this requires approximately 8% more bits.

4.2 Algorithm overview

The values for the offset field and the bank bit are set at cache miss time, as Ops are received into the cache-memory interface. Since the fields are computed and set as the cache block is filled, no extra cycles are required. There are cases when these fields are set during a cache hit, but these are exceptional cases and are explained later. The basic operation of the algorithm is outlined in the following steps.

1. **Fetch the missing MultiOp:** When a cache miss occurs, Op fetches are generated to the next level of the memory hierarchy. All fetch requests are performed as a multiple of the block size; that is, given a block size n, all fetch requests are either for n or $2n$ Ops. Since the size of the missing MultiOp is unknown at miss time, enough Op fetches are generated to retrieve the largest possible MultiOp. If the miss address

begins on a cache block boundary (offset 0 in a block), n Ops fetches are generated, to fetch the entire cache block. If the miss address is at offset $0 < m < n$, $2n$ fetches are generated to fill the first and second blocks, in case the MultiOp spans two blocks. Note that this prefetches at least some Ops of the *next* sequential MultiOp (the one that follows the MultiOp being currently requested).

2. **Set the offset field and the bank bit:** A counter in the cache-memory interface is used to count the number of Ops per MultiOp. When a fetch is initiated, the counter value is set to zero. For every subsequent Op that does not have a set tail bit, the counter is incremented by one. When an Op with a set tail bit is received, the counter is reset to one (what is done with the counter value before resetting is explained shortly). Again, for every subsequent Op that does not have a set tail bit, the counter is incremented by one. When an entire MultiOp has been retrieved, the counter value and the starting offset m of the just-retrieved MultiOp are used to determine the starting offset p for the *next* MultiOp. If $p > m$, the bank bit is set to 0, since the next MultiOp begins in the same cache block as the just-retrieved MultiOp. If $p \leq m$, the bank bit is set to 1.

   (a) If the starting offset of the missing MultiOp is non-zero, then the first Op received by the cache-memory interface could have an unset header bit. In this case, it and all Ops up to the first Op with a set tail bit are the latter portion of a MultiOp. The value of the counter when the first Op with a set tail bit is received is only a *partial length*. The partial length is stored in the first offset field of the cache block, and the offset field's valid bit is set to invalid.

   (b) Due to prefetching, Ops at the *end* of a fetched cache block might be the beginning portion of a MultiOp. If all of the Ops for this MultiOp are not retrieved during the current fetch (an Op with a set tail bit is not the last Op in the block), the counter value at the end of the fetch holds only a partial length for the MultiOp. Similar to the case described above, the partial length is stored in the last offset field for the cache block, and the offset field's valid bit is set to invalid.

5 Illuminating the algorithm

5.1 The Basic Case

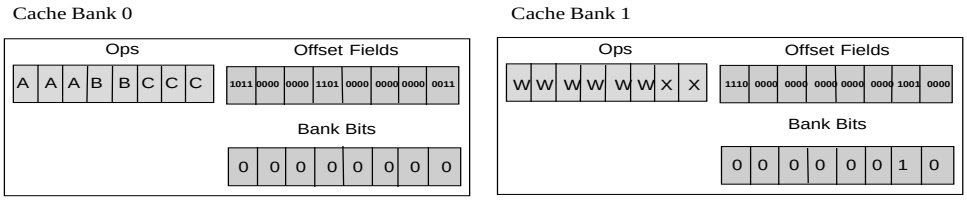
The algorithm is best explained through examples. In Figure 5(a), the cache state is shown when a miss occurs for MultiOp C. The miss is detected by the presence of an invalid offset field for the location accessed. Since the location accessed is at offset five, enough Op fetches are generated to fill to the end of the next cache block. Not only is the remainder of MultiOp C fetched but so is at least part of MultiOp D, as shown in Figure 5(b). Only a portion of D is fetched, so the last offset field in its block is set as a partial length.

5.2 Invalidations

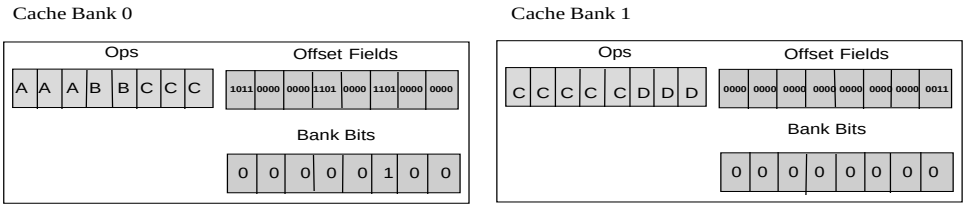
When a MultiOp is loaded on a cache miss, it might displace portions of other MultiOps. If the *beginning* of a MultiOp Z is replaced, this is detected when Z is later accessed and a tag mismatch occurs. This case is shown in Figure 5(b) when MultiOp X is displaced. When only the *latter portion* of a MultiOp Z is replaced, a situation can occur where the beginning of Z is in the previous block with a valid offset field indicating that Z continues into the successor block. This scenario is shown in Figure 6 and is handled by performing extra work at cache miss time. In Figure 6(a), MultiOp C is shown as spanning two cache blocks. When the miss for W is generated and the cache fill for W commences, the latter Ops of MultiOp C are displaced although the beginning portion of C is still cache resident. To indicate that all of C is not cache-resident, the offset field for C is marked invalid. This *invalidation* process requires searching the offset fields for C's offset entry. This search can be performed in parallel with the cache fill and so does not require any extra cycles. During a subsequent access for MultiOp C, its invalid offset field indicates that all of C is not resident in the cache, and a fetch is generated.

5.3 Ghost blocks

An interesting situation can occur when an entire MultiOp is cache-resident and spans two blocks but its offset field is invalid. In this case, the cache blocks are referred to as *ghost blocks*. Figure 7 depicts how this situation can occur. When a cache miss occurs for MultiOp A, the beginning of MultiOp C is prefetched, as shown in Figure 7(b). Also, a partial

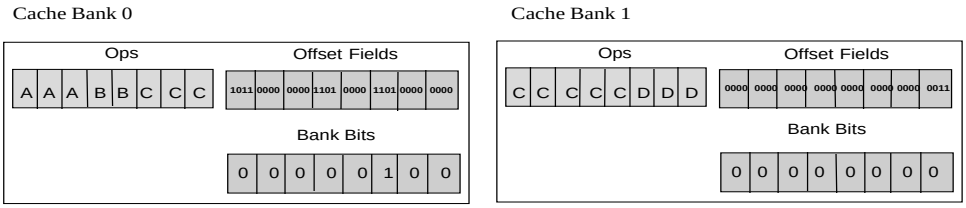


a) Cache blocks before fetching MultiOp C

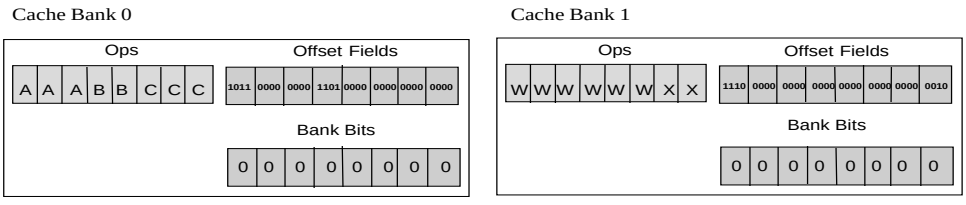


b) Cache blocks after fetching MultiOp C and prefetching part of D

Figure 5: Fetching due to a cache miss: A miss for MultiOp C is shown and the states of the data array, offset fields, and bank bits are shown for when the miss is generated (a) and after MultiOp C has been fetched (b). Part of MultiOp D was also fetched, due to prefetching. All of D was not prefetched so the corresponding offset field is set to invalid.

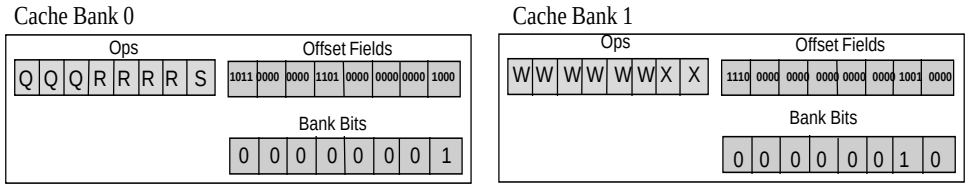


a) Cache blocks before fetching MultiOp W

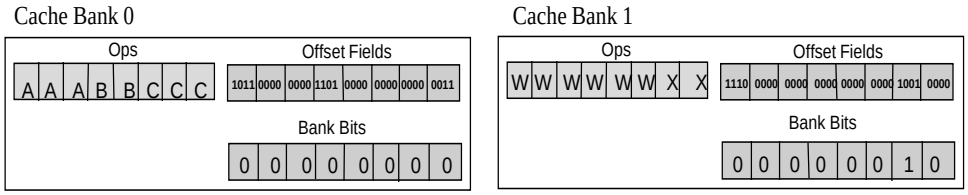


b) Cache blocks after fetching MultiOp W and prefetching part of X

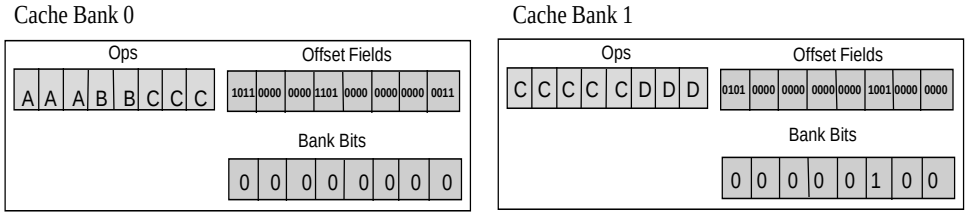
Figure 6: Invalidations of displaced MultiOps: A miss for MultiOp W is shown and the states of the data array, offset fields, and bank bits are shown for when the miss is generated (a) and after MultiOp W has been fetched (b). Part of MultiOp X was also fetched, due to prefetching. All of X was not prefetched so the corresponding offset field is set to invalid.



a) Cache blocks before fetching MultiOp A



b) Cache blocks after fetching MultiOp A and prefetching B and part of C



c) Cache blocks after fetching MultiOp D and "prefetching" the latter part of C

Figure 7: Ghost blocks: MultiOp C is shown as being fetched into the cache yet having an invalid offset field. The beginning portion of C is prefetched and a partial length is assigned (b). The latter portion of C is then prefetched and a partial length assigned for that part also (c). If C is later accessed, an extra cycle is required to detect that although C's offset field is invalid, it is entirely cache-resident and its offset field and bank bit need to be set.

length for C is set. A later cache miss for MultiOp D causes a cache miss, and D is fetched and the latter portion of C is prefetched, as shown in Figure 7(c) (assume that the beginning portion of C has not been displaced from the cache and an access for C has not occurred before the access for D) [1]. All of the Ops for MultiOp C are now in the cache but because of the access pattern of the program, C's offset field is not set to valid. When an access for C is generated, the tags for both blocks match, indicating that C is cache-resident, but the offset field is invalid, indicating that C's length has yet to be computed. An extra cycle is required to detect that two partial lengths for C are set and then set the offset field and bank bit for C. MultiOp C can then be forwarded to the next stage in the pipeline.

[1] All of the Ops that compose MultiOp D are not shown, as D spans two cache blocks. Assume that they were also fetched into the cache.

6 Implementation Considerations

The use of dedicated hardware for NextPC computation may appear complex at first glance but in fact the hardware requirements are straightforward. Where Figure 4 gives a high-level view at the hardware, Figure 8 provides details of how the logic could be implemented. In actual implementation, two possible values of the `tag+index` for NextPC are computed (Steps 1c and 1d in Figure 8). The reason for this is because of the access latency of the offset field and bank bit storage (henceforth referred to as the length info storage). (In the following discussion, `tag+index` for the current PC is referenced as $\mathtt{PC_{tag+index}}$, and `tag+index` for NextPC is referenced as $\mathtt{NextPC_{tag+index}}$.) Note that $\mathtt{NextPC_{tag+index}}$ can have one of two values. If the bank bit is 0, $\mathtt{NextPC_{tag+index}}$ is

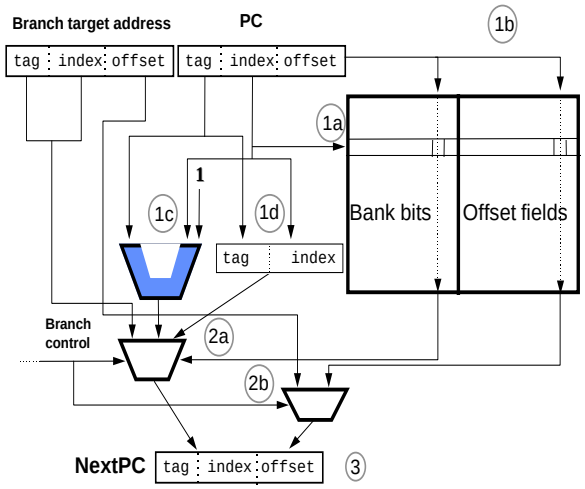


Figure 8: A possible implementation of the NextPC logic. Parallel row and column access is performed into the storage for the offset fields and the bank bits (Steps 1a and 1b). In parallel with this, `tag+index` for *two* potential NextPCs are computed, in the case where the bank bit turns out to be 1 (Step 1c) and the case where it is a 0 (Step 1d). After the access of the bank bit storage is complete, the bank bit and the branch control line are used to multiplex between the two potential `tag+index` quantities and high order bits from any incoming branch target address (Step 2a). A parallel multiplex operation controlled by the branch control line selects the correct `offset` (Step 2b). The final step is concatenating the correct `tag+index` and `offset` (Step 3).

the same as $\mathsf{PC}_{\texttt{tag+index}}$. If the bank bit is 1, $\mathsf{NextPC}_{\texttt{tag+index}}$ is generated by adding 1 to $\mathsf{PC}_{\texttt{tag+index}}$. So, potentially an addition is required to generate NextPC. If the add is started after the access of the length info storage, it is performed at the very end of the clock cycle. This can lengthen the time required for NextPC computation significantly. An alternative is to generate both possible $\mathsf{NextPC}_{\texttt{tag+index}}$ values and to start doing so in parallel before the length storage info access completes. The addition is started at the beginning of the clock cycle, just after $\mathsf{PC}_{\texttt{tag+index}}$ is available and before the length info storage access has completed. The other possible $\mathsf{NextPC}_{\texttt{tag+index}}$ value is fetched from the PC. After the length info storage access has completed, the bank bit is used to select the correct new $\mathsf{NextPC}_{\texttt{tag+index}}$ value to use. The tradeoff is that a two-way multiplex function instead of an add is performed after length storage info access.

NextPC computation requires an access to the length info storage. This access if performed in parallel with cache access (timing issues are examined in Section 7.2. If the time required for length storage info access is the same as for cache access, NextPC generation as described above would stretch the cycle time, because of the extra multiplexing step. However, the logic can be designed so that cycle time is unaffected. The length info storage can be implemented as a separate storage that is substantially smaller than the cache. Hence, it can be accessed quickly as a two-dimensional memory, with row and column select performed in parallel [17][p.151]. The add is performed in parallel with the access and does not stretch the cycle time as it operates on narrower operands than an integer add by the ALU. The key step then is the multiplexing performed at the very end. In fact, this multiplexing is required regardless of the selection of potential $\texttt{tag} + \texttt{index}_{\mathsf{NextPC}}$ values. A multiplexor is required so that a branch target address can be selected when a branch Op executes. Such a multiplexor is required in any machine that supports branch operations. The multiplexor that selects between the two NextPC `tag+index` quantities can be combined with the simple multiplexor that selects between NextPC and a branch target address. For the banked cache design, two multiplexors are required, to select the proper high order bits (`tag+index`) and the proper low order bits (`offset`). A control line from the FU that executes branch Ops is used as a select line into both multiplexors, as shown in Steps 2a and 2b in Figure 8. A larger amount of logic is required for NextPC computation in a banked cache as compared to a traditional cache. As most of the additional logic operates in parallel, the cycle time requirements are no more demanding than for a traditional design.

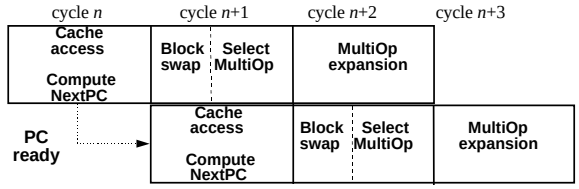
7 Related Issues

7.1 Sub-blocking and computing NextPC

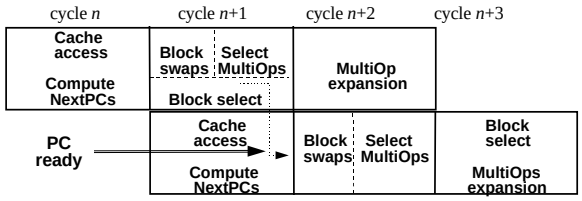
The banked cache design can employ sub-blocking although this may be of limited benefit [2]. NextPC computation in the presence of sub-blocking is performed exactly as it is when sub-blocking is not used. An offset field and a bank bit are maintained for every Op in every cache block and are used in the same fashion as before.

7.2 Set associativity

NextPC computation is more complicated when set-associativity is used. Firstly, set-associativity might not be appropriate for a cache design that uses a hit-path expander. Set associativity requires parallel tag compares to select the correct block from the set. As the parallel tag compares are performed, each *pair* of blocks might also need to be swapped *in parallel*, so that the correct MultiOp is ready to be sent to the expander as soon as the block select stage is complete. Additionally, the block select function requires a stage of multiplexing that is not needed in a direct-mapped cache, which can stretch the cycle time. For these two reasons, high degrees of associativity may not be a good choice for the banked cache design.



a) Pipeline stages for a direct-mapped organization



b) Pipeline stages for a set-associative organization

Figure 9: Pipeline stages for NextPC computation.

If set associativity is used, NextPC computation must still be performed in a single cycle. An intuitive approach might be to compute multiple "candidate" NextPCs in parallel and then select the proper NextPC based on the block select. This provides NextPC at too late a time, however. The problem is depicted in Figure 9. In a direct-mapped cache, only one set of (offset field, bank bit) is accessed. There is no choice involved as to which is the proper NextPC to use, and NextPC is available at the beginning of the next clock cycle (n+1) for the next cache access (Figure 9(a)). In a set-associative cache, multiple NextPCs are computed, and these candidates are available at the same time as NextPC in a direct-mapped cache (the beginning of cycle n+1). However, because the block select is performed in the *next* pipeline stage in cycle n+1, the correct NextPC to use is not known until one cycle later, at the beginning of cycle n+2 (Figure 9(b)). This is unacceptable because NextPC must be available at the beginning of cycle n+1. One solution to this problem is to predict in cycle n which block will be selected by the block select in the next stage (cycle n+1), a technique termed *way prediction*. Given the complexities associated with set-associativity for the banked cache, a technique that works well at low associativities is perhaps best suited. Juan *et al.* proposed a design that performs way prediction for a two-way set-associative cache in a single cycle [18]. This technique can be adapted for NextPC computation for a set-associative banked cache. The use of way prediction with the banked cache is a future area of research within the TINKER group.

8 Conclusion

The instruction fetch requirements for VLIW architecture demand high bandwidth to the instruction cache. When using a compressed encoding for the instructions, one alternative for an instruction fetch mechanism is to use a banked instruction cache. Due to the variable length nature of compressed encoded instructions, on every cycle the address for cache access for the subsequent cycle – NextPC – must be computed. This paper presented a scheme for NextPC computation using an offset field and a bank bit for every Op stored in the instruction cache. The algorithm to set and use these fields was also outlined. Examples were used to illustrate the nuances of the algorithm, and implementation issues regarding the feasibility of the hardware were discussed. Further work is being performed to examine alternative encodings and instruction fetch support for VLIWs.

Acknowledgments

We are grateful to Professor Wentai Liu of North Carolina State University for his discussions with us on the implementation of memory structures.

References

[1] 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.

[2] 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. International Symposium on Microarchitecture*, (Paris, France), Dec. 1996.

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

[4] 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.

[5] 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), pp. 153–161, Jan. 1995.

[6] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *Computer*, vol. 22, pp. 12–35, Jan. 1989.

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

[8] 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. Third Int'l. Conf. on Architectural Support for Prog. Lang. and Operating Systems.*, (Palo Alto, CA), pp. 180–192, Oct. 1987.

[9] 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*, (San Diego, CA), pp. 60–66, Dec. 1988.

[10] M. Smotherman and M. Franklin, "Improving CISC Instruction Decoding Performance Using a Fill Unit," in *Proc. 28th Ann. International Symposium on Microarchitecture*, (Ann Arbor, MI), pp. 313–323, Dec. 1995.

[11] D. B. Papworth, "Tuning the Pentium Pro microarchitecture," *IEEE Micro*, vol. 16, pp. 8–15, Apr. 1996.

[12] D. Christie, "Developing the AMD–K5 architecture," *IEEE Micro*, vol. 9, no. 2, pp. 16–26, 1996.

[13] D. Draper *et al.*, "A 93 mhz, x86 microprocessor with on–chip L2 cache controller," in *Proc. 1995 International Solid-State Circuits Conference*, (San Francisco, CA), pp. 172–173, Feb. 1995.

[14] 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*, (Pittsburgh, PA), pp. 2–9, June 1987.

[15] D. Alpert and D. Avnon, "Architecture of the Pentium microprocessor," *IEEE Micro*, vol. 13, pp. 11–21, June 1993.

[16] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel, "Optimization of instruction fetch mechanisms for high issue rates," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, (Santa Margherita Ligure, Italy), pp. 333–344, June 1995.

[17] B. Prince, *Semiconductor memories*. West Sussex, England: John Wiley & Sons, Ltd., 2 ed., 1991.

[18] T. Juan, T. Lang, and J. J. Navarro, "The difference–bit cache," in *Proc. 23nd Ann. International Symposium Computer Architecture*, (Philadelphia, PA), pp. 114–120, May 1996.