© Copyright by Matthew Martin Crum, 2001
DESIGN ALTERNATIVES FOR CACHING LONG REGIONS
OF THE DYNAMIC INSTRUCTION STREAM

BY

MATTHEW MARTIN CRUM

B.S., Valparaiso University, 1999

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2001

Urbana, Illinois
ABSTRACT

Noncontiguous control flow challenges high-bandwidth execution in microprocessors by prematurely terminating a fetch to less than a full fetch width. To deal with this problem, methods have been devised ranging from branch prediction schemes to compiler techniques for reducing taken control flow to hardware mechanisms for caching dynamic traces from the instruction stream. Recently, a technique to form long instruction sequences called frames using branch promotion has been proposed. Frames are instruction entities that can grow to be very long and must be cached as atomic units.

Designing a cache for these long instruction regions involves evaluating many complex alternatives. In this thesis, the design space of such a cache is explored. The contributions of this thesis include: (1) an evaluation of several standard parameters for frame cache design, such as size and associativity; (2) an evaluation of two different basic design options for the frame cache structure; and (3) an understanding of the frame cache's behavior over metrics such as the average lifetime of a cache entry, average utilization of space in the frame cache, and specific implementation details concerning access ports. The data and conclusions presented here can be applied to other contexts, including trace cache designs.
To Susan, for her incredible patience.
ACKNOWLEDGMENTS

Thanks to Satarupa Bose, Brian Fahs, Brian Slechta, and Tony Tung for providing feedback and rePLay simulation support. Their contributions made this paper possible.

Special thanks to Francesco Spadini for the countless times he allowed me to pick his brain for ideas and constructive criticisms.

Thanks to the corporate sponsors of the ACS group at Illinois: AMD, Intel, and HP.

And finally, thanks to Sanjay Patel, whose advice, technical direction, and encouragement have made my graduate school experience challenging and educational.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MOTIVATION</td>
<td>1</td>
</tr>
<tr>
<td>2 THE FRAME CACHE SYSTEM</td>
<td>4</td>
</tr>
<tr>
<td>3 MULTILINE CACHING SCHEMES</td>
<td>7</td>
</tr>
<tr>
<td>3.1 Chaining Scheme</td>
<td>7</td>
</tr>
<tr>
<td>3.2 Pooling Scheme</td>
<td>10</td>
</tr>
<tr>
<td>3.3 Implementation Details</td>
<td>13</td>
</tr>
<tr>
<td>3.3.1 Locking</td>
<td>13</td>
</tr>
<tr>
<td>3.3.2 Wait queue</td>
<td>14</td>
</tr>
<tr>
<td>3.3.3 Replacement policies</td>
<td>16</td>
</tr>
<tr>
<td>3.3.4 Evictions</td>
<td>16</td>
</tr>
<tr>
<td>3.3.5 Feedback to frame constructor</td>
<td>17</td>
</tr>
<tr>
<td>4 SIMULATION ENVIRONMENT</td>
<td>18</td>
</tr>
<tr>
<td>4.1 Benchmarks</td>
<td>18</td>
</tr>
<tr>
<td>4.2 Simulation Environment</td>
<td>19</td>
</tr>
<tr>
<td>4.3 Baseline Processor</td>
<td>19</td>
</tr>
<tr>
<td>5 PERFORMANCE ANALYSIS</td>
<td>21</td>
</tr>
<tr>
<td>5.1 Frame Cache Associativity Analysis</td>
<td>21</td>
</tr>
<tr>
<td>5.2 Frame Cache Size Analysis</td>
<td>24</td>
</tr>
<tr>
<td>5.3 Performance Numbers for Chaining vs. Pooling</td>
<td>25</td>
</tr>
<tr>
<td>5.4 Frame Entry Lifetime Analysis</td>
<td>26</td>
</tr>
<tr>
<td>5.5 Frame Cache Writes</td>
<td>31</td>
</tr>
<tr>
<td>5.6 Frame Cache Reads</td>
<td>33</td>
</tr>
<tr>
<td>5.7 Wait Queue Sensitivity Analysis</td>
<td>35</td>
</tr>
<tr>
<td>5.8 Replacement Policies</td>
<td>36</td>
</tr>
<tr>
<td>6 FRAME CACHE UTILIZATION</td>
<td>37</td>
</tr>
<tr>
<td>6.1 Chaining Scheme Utilization Analysis</td>
<td>37</td>
</tr>
<tr>
<td>6.2 Pooling Scheme Utilization Analysis</td>
<td>39</td>
</tr>
<tr>
<td>6.3 Further Comparison of Chaining and Pooling</td>
<td>41</td>
</tr>
<tr>
<td>7 A GLANCE AT REDUNDANCY</td>
<td>46</td>
</tr>
<tr>
<td>8 CONCLUSION</td>
<td>48</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>50</td>
</tr>
</tbody>
</table>
CHAPTER 1

MOTIVATION

Optimal performance in highly superscalar processors requires that as many instructions be fetched each cycle as can be potentially consumed by the execution engine. Control flow, in particular taken control flow, challenges this basic need for high-bandwidth execution by prematurely terminating a fetch to less than a full fetch width. Many schemes have been proposed to deal with this important limitation to higher performance, ranging from compiler techniques to reduce the incidence of taken control flow [1, 2], to hardware mechanisms for caching traces [3, 4, 5], to combined approaches involving both compiler transformations and hardware [6, 7].

More recently, a technique has been proposed that extends previous trace caching approaches by generating atomic sequences of instructions from the dynamic instruction stream [8] longer than sequences investigated in previous trace cache work. These atomic regions, called frames, have the property that if any one instruction in the region executes, then they all execute. Coupled with the technique to construct these dynamic sequences is a hardware recovery mechanism to undo the execution of a frame if an assertion were to be incorrect (known as firing). By identifying highly biased branches and converting them into control flow assertions, the techniques proposed in [9] are able to create frames of over 100 instructions. As a whole, these frames can span 80% of the dynamic instruction stream (the remainder of the stream is covered by regular basic blocks), with the probability of assertion firing being very low (approximately 3%).
While addressing the performance limitations caused by taken control flow, frames potentially offer another dimension to boost performance. Their atomic nature and long length make them attractive candidate regions for dynamic optimization. The hardware recovery mechanism supports this optimization by undoing a frame whenever the assumptions for its optimization do not hold during execution.

For microarchitectures that dynamically create such long atomic regions, the rePLay Framework for example, caching these long instruction entities becomes an important and challenging problem. Just as with traces, they must be cached as a singular, atomic entity. The traditional approach to trace cache design has been to store each trace entirely in one cache line. Because of their longer and wider range of lengths, this approach is not an effective solution for frame cache design. The traditional approach would either inhibit the construction of longer frames or would result in inefficient utilization of cache space on frames smaller than the cache line size.

In contrast, storing these traces or frames over multiple cache lines results in the region consuming cache space in proportion to its size. For example, a frame of size 80 instructions would span ten 32-byte cache lines (assuming an instruction occupies 4 bytes). With this technique, cache line size does not limit region size nor lead to ineffective use of cache space. The essential problem with caching these dynamic instruction regions over multiple cache lines is that the lines must be treated as a unit. If a portion of a frame or trace is to be evicted because of a conflict for a particular cache line, then the entire trace or frame must be evicted. Devising a cache to provide this functionality presents interesting design options. In this thesis, two options are investigated, one called chaining and another called pooling, for designing a cache that can accommodate atomic blocks of instructions of varying lengths.

This problem and its potential solutions are not exclusive to frame systems such as rePLay. Trace cache schemes that do not limit trace size to one cache line, such as a patent filed by Krick et al. at Intel [10], or dynamic translation mechanisms that generate optimized code sequences [11], can potentially make use of instruction caching structures that allow atomic entities of various sizes to be cached.
The contributions of this thesis include: (1) an evaluation of several parameters for frame cache design; (2) an evaluation of two different basic design options for the frame cache structure; and (3) an understanding of the frame cache’s behavior over metrics such as the average lifetime of a cache entry, average utilization of space in the frame cache, and specific implementation details concerning access ports. Chapter 2 gives a discussion of the rePLay Framework, which is the frame processor system on which all experiments are performed. Chapter 3 contains a detailed description of the two frame cache schemes, chaining and pooling, and offers a discussion of specific implementation details. Chapter 4 provides specific details about the simulation environment. Chapter 5 provides an analysis of the performance and behavior of each frame caching scheme. Chapter 6 contains a discussion of the topic of frame cache space utilization and an explanation for why optimal space utilization is difficult. Chapter 7 offers a surface-level investigation into the information redundancy that potentially exists in a frame cache. Finally, Chapter 8 offers some concluding reflections on the work presented in this thesis.
CHAPTER 2

THE FRAME CACHE SYSTEM

In this chapter, the microarchitecture that is the basis for evaluation is described. Called the rePLay Framework, this hardware model creates frames from the dynamic instruction stream by monitoring branch behavior and caches them in the processor’s fetch engine. Since frames are atomic entities that have a wide distribution of lengths, the rePLay Framework requires a multiline caching mechanism.

A high-level diagram of the rePLay mechanism is shown in Figure 2.1. Frames are formed by the frame constructor after instructions commit results to architectural state. The frame constructor promotes highly biased branches into assertion instructions. An assertion checks a branch’s condition without actually causing control flow redirection. The use of branch promotion removes the control flow dependencies from frames and allows them to be treated as single-entry, single-exit, atomic regions of code. This atomicity of frames enables dynamic optimizations to be performed with regard only for data dependencies [8, 9].

![Diagram]

Figure 2.1: The rePLay Framework
The frame constructor keeps growing a frame until a branch is reached that cannot be promoted into an assertion. At this point, the frame is passed to a dynamic optimization engine that performs low-level code optimizations such as specialization, dead code removal, and common subexpression elimination. Optimized frames are passed from the optimization engine to the frame cache in the front-end to be stored for use by the execution core. Note that in this thesis the optimization engine is turned off.

The frame cache is accessed using path history and the next block address output of a branch predictor. Path history is a recording of the last N basic block addresses, where N is a design parameter. Path history is the context in which a frame is constructed and therefore is used by the front-end to determine when a particular frame should be fetched. For example, a particular frame consisting of the blocks ABCDE might be constructed if the path leading to this frame is XYZ. In this way, XYZ forms the context of the frame ABCDE. The fetch mechanism will fetch this frame only if the current path history is XYZ and the predictor determines that the next target is A.

The logic needed to accommodate this context-based fetch is called the sequencer. A diagram of this logic is shown in Figure 2.2. In each fetch cycle, the fetch engine attempts to fetch from both the frame cache and the instruction cache. If there is a hit in the frame cache, the sequencer chooses to issue those instructions to the execution core. Otherwise, instruction fetch proceeds from the normal instruction cache and memory hierarchy.

This mechanism bears similarity to proposed mechanisms for trace caching. One should keep in mind that trace caches are primarily used for boosting the fetch and decode bandwidth of a wide-issue processor. Frame caching supports this objective but has the additional goal of providing a stronger basis for dynamic customization of the application’s instruction stream to its run-time behavior. The difference between a trace cache mechanism and the frame mechanism is largely in the way the two are delineated. Traces are formed once a certain number of branches \[3, 12\] (based on branch predictor bandwidth) or instructions (typically tied to the processor’s fetch width, and therefore usually cache line size) is reached. Frame construction, on the other hand, terminates frames at unbiased (nonpro-
Figure 2.2: A pipelined frame sequence mechanism

Notable) branches, thereby giving frames (1) the property of atomicity and (2) the potential for long lengths. It is because of these two properties that a frame cache must adopt a multiline policy for caching frames. In the next chapter, two mechanisms that provide such a policy are described.
CHAPTER 3

MULTILINE CACHING SCHEMES

The frame cache is expected to store as many frames as possible at one time, retain useful frames, and maintain the atomicity of its entries. Thus, careful decisions must be made in its design to maximize these goals.

The following sections provide discussion of two ways in which the frame cache can be organized for multiline storage. The first method is referred to as the chaining scheme. It allows freedom for any part of a frame entry to go to any block in the cache. The second method is called the pooling scheme. It has two separate storage structures, one designated for a frame’s first segment of instructions (called head segments) and the other in which subsequent frame segments (or body segments) are stored. Both schemes are evaluated through several sensitivity studies in order to determine which is more effective and under what situations.

3.1 Chaining Scheme

The chaining frame cache is an N-way set-associative cache structure. It has a line size of \( L \) instructions equal to the instruction fetch width of the underlying processor. Thus, a frame containing \( F \) instructions is stored in the frame cache using \( \text{CEILING}(F/L) \) cache lines. As an example, storing an 80-instruction frame requires 10 cache lines, assuming each cache line can store eight instructions. However, storing an 81-instruction frame requires 11 cache lines.
Storing a frame over multiple cache lines in the chaining scheme requires that each L-instruction segment of a frame have some way of being identified as belonging to a particular frame entry. Furthermore, the proper sequential order among a frame entry’s segments must be maintained. In the chaining scheme, each cache line has pointer bits used to maintain this order. Each line also has a bit indicating whether or not the segment in that line is a frame entry’s head segment or a subsequent body segment. Figure 3.1 illustrates how a frame is stored in the chaining scheme. Each cache line has a tag (the frame’s start PC), the frame segment’s instructions, bits to indicate valid, head and tail segment, and a pointer to the next frame segment. The pointer fields need only be indexes into the cache structure.

![Frame Cache Diagram](Image)

**Figure 3.1**: An illustration of the chaining scheme

Writing into the chaining frame cache is a multicycle operation. In the first cycle, a frame’s path history, or context, is used as the hash key to determine in which set the head segment of frame instructions belongs. The particular cache line within the set can be chosen using a variety of schemes. Random selection is used here, with a preference given to invalid lines. In other words, within a given set, invalid (empty) cache lines are targeted first, followed by a random selection if no lines are empty. In this and subsequent write cycles, two basic scenarios are possible for each segment of a frame: (1) the line is invalid or (2) the cache line being overwritten belongs to an already valid frame entry.
In the former case (the line is invalid), the current frame segment is simply written into the invalid entry. In order to determine where to write the next segment of the new frame, a random selection is done among all sets in the frame cache. Then a line within that set is chosen based on random selection with a preference for invalid cache lines, as described above.

In the latter case (the line is occupied), the old entry’s space is used to store the new entry. Note that the entire frame that the old entry corresponds to must be invalidated. To do otherwise would result in a fragmented frame in the frame cache, thereby violating the atomicity of the frame, potentially making the instructions within the partial frame meaningless. To accomplish this invalidation, subsequent writes of the new frame will follow the forward pointers in each entry of the old frame in order to overwrite the old frame with the new one. Note that the last entry of the old frame will point to the head entry (thus the circular nature of the chaining).

Each segment of a frame is written into the cache until the entire frame is written, at the rate of one frame segment per cycle. If the incoming frame is smaller than the frame being overwritten, then the remaining lines of the old frame must be invalidated at the rate of one per cycle. The tail bit of the last block occupied by the new frame is set, and the forward pointer bits are set to wrap around to the head of the new frame. In this way, subsequent entries that overwrite this frame can follow the pointers to wrap around and use the whole space stored by this frame. Note that there must be some way of keeping track of the segment that the new entry started overwriting in the old chain so that the cache controller knows when the old chain’s space is exhausted.

Reading from the chaining frame cache is also a multicycle operation. In the first read cycle, the path history and the next address prediction of the branch predictor are used to access the frame cache. The predicted address is used as the tag compare with the frame’s start PC. If a hit occurs, the head segment of the desired frame is read out, along with the pointer to the next body segment of the frame. In subsequent read cycles, the pointer bits are followed until the entire frame is read from the frame cache. The read proceeds at the
rate of one segment per cycle (thus, the size of each cache line is set to be one fetch-width worth of instructions).

While a frame is being read from the cache, none of the cache lines occupied by that frame can be overwritten. This is required in order to maintain the atomic nature of the frame. If a portion of a frame is overwritten as the frame is being read, then a potential error situation may arise. In order to prevent this situation, a policy could be adopted in which a write to the cache cannot occur during read cycles. However to do so would be costly to performance. Instead, a lock register is maintained such that lines belonging to the current frame being read are prevented from being overwritten. When a frame read is initiated, the start PC of the frame is written into the lock register. Lines that match this lock register will not be selected for any write that occurs concurrently during the read. Section 3.3.1 discusses the concept of locking in more detail.

This general scheme can be applied to trace caches as well, in particular for ones in which traces are permitted to span multiple cache lines. Intel has patented a trace cache design [10] similar to this chaining scheme. In the patent document, they describe a write scheme in which subsequent segments of a trace entry are written into physically contiguous sets within the cache.

3.2 Pooling Scheme

The pooling frame cache is comprised of two structures. The first structure, called the head partition, is an N-way set-associative cache structure reserved for head segments of frame entries. It has a line size of L instructions equal to the instruction fetch width of the underlying processor. The other structure, called the body partition, is a direct-mapped cache reserved for body segments of frame entries. Figure 3.2 on the next page illustrates how a frame is stored in the pooling scheme. Each head cache line has a tag (the frame’s start PC), the frame’s instructions, a bit to indicate valid, and a pointer into the body partition to indicate where the next segment of the frame resides. Body lines only need to contain the
corresponding instructions and a pointer to the next body segment of the frame. Coupled
with the body partition is a mechanism to indicate which lines are currently invalid. This
mechanism is called the freelist.

![Frame Cache Diagram](image)

Figure 3.2: An illustration of the pooling scheme

The pooling scheme requires a balance between the number of head blocks in the head
partition and body blocks in the body partition. For a cache of a particular size, say 4K
cache lines, some number of those cache lines must be allocated to the head partition and
the remainder to the body partition. The number of lines in the head partition limits the
total number of frames that can be simultaneously stored in the frame cache.

Writing into the pooling frame cache involves hashing into the proper set in the head
partition and selecting a line based on some selection process. Random selection with a
preference for empty cache lines is used here. Subsequent body segments of the frame are
written into the body partition by querying the freelist controller for the next available cache
line. If the freelist is empty, then a block from the head partition must be selected in order
to return some blocks to the freelist. Again, this selection is done randomly, this time with
a preference for frames spanning multiple cache lines (those that own body cache lines).
In other words, the next set is chosen randomly, and a line within that set that actually
owns body cache lines is chosen. Choosing a head cache line that is not attached to any
body blocks is a useless selection because it does not have any body lines to return to the
freelist. In essence, the newly invalidated body blocks can be overwritten by the pending
frame entry’s remaining body segments until either the frame entry is completely written or another frame must be invalidated. The selection of a new frame for invalidation causes a one-cycle write stall. The remaining unused blocks from an invalidated entry are returned to the freelist.

Reading from the pooling frame cache is essentially the same as for the chaining scheme. After a read hit, the first cycle of the access involves reading the head segment, and pointer bits are followed in subsequent cycles to read out the remaining frame segments.

The read/write overlap problems that occur in the chaining scheme become simplified in the pooling scheme. This is due to the fact that the pooling scheme’s interface to the rest of the system exists only at the head partition, which makes every frame look equally long to all other frames. The replacement and eviction policies can only target head blocks in the pooling scheme, which means that the potential for writing into the middle of an old frame entry in the chaining scheme is removed in the pooling scheme. This alleviates many write conflicts and enables writes and reads to overlap without extensive control logic to manage the locking.

The pooling scheme also requires a mechanism to maintain a list of free cache lines in the body partition. While various design alternatives exist here, a rather simple one is proposed. The freelist itself is a chain of nodes through the body partition and is represented as two pointers, one for the head of the freelist and one for the tail. Whenever a frame is evicted, its list of body segments (i.e., the pointer in the frame’s head segment), is added to the freelist by updating the freelist tail cache line to point to the evicted frame’s head segment. The freelist tail pointer then is updated to point to the evicted frame’s tail segment. To facilitate this, each cache line in the head partition can be augmented to store tail pointers for the frames it contains.

An alternative implementation of the pooling scheme would be to store pointers to all body segments of a frame in the head segment’s cache line. In other words, the head cache line would act as a map pointing to all body segments of the frame entry. All body segment pointers would be available in the first cache access rather than one per subsequent access
in the scheme described above. This kind of organization may have certain advantages. For instance, having pointer bits to body segments in the head partition could facilitate a kind of redundancy compression by allowing several frame entries to point to the same eight-instruction body segment. This is similar to the block-based trace cache discussed in [13].

While there may be advantages to having all of the body segment pointers available in the first access instead of having to follow a linked list of frame segments, there are some drawbacks. The most significant of these would be the complications involved in maintaining the freelist. To invalidate body partition cache lines and add them back to the freelist would require more expensive cache control logic since the freelist would no longer be able to exist as a linked list with simple head and tail pointers. Also, the fact that each head cache line would need to have enough pointer bits to facilitate storing the maximum frame size may lead to wasted chip space, whereas the scheme proposed above only has as many pointer bits as there are body partition cache lines.

3.3 Implementation Details

3.3.1 Locking

As mentioned in Section 3.1, a locking mechanism must exist to deal with potential read/write overlaps in the frame cache. Since frames are atomic entities that require multiple cycles to read, it is required that the frame being read does not get overwritten before it has been entirely read. A scenario can exist in which an incoming frame on the write port either has the same start PC and hashes into the same set as the frame being read, or the cache controller selects a line for overwriting that is currently occupied by a frame being read. To protect against this situation, the cache lines currently owned by a frame being read must be locked, meaning they cannot be overwritten during the read. This ensures atomicity of reading from the frame cache.
One may argue that the frame being read can be overwritten in cache lines which have already been read by the fetch engine as long as the lines of the frame still unread are not overwritten. In general, a read can stay ahead of a write in the frame cache. *Staying ahead* simply means that all cache lines occupied by the frame currently being read are indeed read before being overwritten by an incoming frame entry. However, in the exceptional case of a fetch stall (for instance, when the issue window is full), keeping the reads and writes synchronized becomes difficult. Since writes to the frame cache happen infrequently relative to the total execution time of a program, it is easier to lock an entire frame until it is completely read. The simple lock register mechanism mentioned in Section 3.1 is the one used in this thesis.

A question may arise in the reader’s mind concerning the need for both read and write access ports. One may argue that having only one port to be used for both reading from and writing to the frame cache is sufficient for performance and would prevent the need for any locking mechanism. This argument fails, however, because an attempt to read from the frame cache is made every clock cycle, which means that writes would be stalled much more often than they are with two access ports and locking. The probability that the frame being written targets a locked entry’s space is low, while the probability that a write would try to take place while the cache is being read at all is much higher. For this reason, having separate read and write ports is quite necessary. Having only one port could lead to unnecessary and costly write delays.

### 3.3.2 Wait queue

Writing into the frame cache is a multicycle operation and thus requires extra care in the design of its write port interface. In particular, a wait queue must exist between the fill unit and the write port. This is needed to handle a scenario in which frames are being sent to the frame cache at a faster rate than they can be written. The length of this queue (in number of pending frames to be written) becomes a design parameter of the frame processor.
The time to write a frame into the frame cache is proportional to its length, as described earlier in this chapter. For instance, a frame of length N instructions would take N/8 cycles to write into an eight-instructions-per-line frame cache. If frames are being presented to the frame cache for writing in intervals less than N/8 cycles, then the wait queue needs to be longer than one entry. Exactly how long to make the queue depends in part on the average frame length in the system and thus the average time to write a frame into the cache.

A more difficult behavior to contend with in designing the frame cache write port interface revolves around the concept of locking. The existence of locking affects the write port’s wait queue design in its potential to stall frame cache writes. If the first element in the queue attempts to access a locked cache line, the write becomes stalled and the wait queue can become backed up. The potential for backup in the frame cache wait queue cannot be fully eliminated. For this reason, a policy must exist to handle the overflow situation. Several policies can be studied. One is to drop pending frames at the front of the queue to make room for incoming frames from the fill unit (rePLay’s frame constructor in the experiments here). Another is to drop incoming frames if no room is available. A third might be simply to drop any frame trying to access locked space in the cache. Another method of handling a frame that seeks a locked entry might be to search the wait queue for other frames to write while the first entry is stalled. However, there would be some cycle penalty for this search. Because of the relative infrequency of locking stalls, this scheme is ruled out. In Section 5.7, it is shown that the scheme chosen has little effect on the performance of the frame cache. Thus, the policy in which frames at the front of the wait queue are dropped when a backup occurs is used throughout the rest of this thesis.

It is also revealed in Section 5.7 that the length of the wait queue has little effect on the overall performance of rePLay. Thus, a wait queue length of 10 is chosen as the baseline,
3.3.3 Replacement policies

Another important design choice to be made is the cache line replacement scheme to use. In the chaining scheme, a policy must be in place to choose a cache line within the set hashed into by an incoming frame’s context, as well as a policy for selecting a new set to overwrite once an old chain has been exhausted. In this thesis, the set is chosen randomly and the line within a set is chosen randomly with a preference for empty cache lines. However, several replacement policies can be used, including least-recently-used (LRU), not-most-recently-used (nMRU), and others.

For the pooling scheme, a policy must exist to select a cache line within the set hashed into by an incoming frame’s context, as well as a policy for selecting a frame entry to invalidate when the freelist is empty. Random selection with a preference for empty cache lines is used in the former case (incoming frame hash) while an occupied head cache line that has body cache lines attached to it is chosen for the latter case (adding to the freelist).

In Section 5.8, the impact of various replacement policies on rePLay performance is observed. The data show that the policies chosen for this thesis, as discussed in Chapter 3, provide adequate performance without the additional control logic and storage space that would be required with a more aggressive policy such as LRU.

3.3.4 Evictions

Another implementation detail is the need for some feedback loop between the execution core and frame cache for maintaining useful frames. Frames are entities formed based on the speculation of control flow direction. When the condition of an assertion instruction within the frame changes from what was speculated in the frame constructor, the assertion fires and architectural state is rolled back to the beginning of the frame.

When an assertion fires, the frame containing it may no longer be useful. Conditions within the program have likely changed to cause the control flow speculation within the frame to be obsolete. Thus, to prevent the now-useless frame from being fetched in subsequent
cycles, the rePLay architecture must be augmented to give the execution core the ability to evict entries from the frame cache.

Currently, when an assertion fires the execution core evicts the frame containing that assertion from the cache. Schemes that allow for some forgiveness to an assertion, such as requiring it to fire twice in a row before eviction, can certainly be studied. Such a study is beyond the scope of this thesis, however.

### 3.3.5 Feedback to frame constructor

A final implementation detail to note concerns the creation of repeated frames in the frame constructor. In order to avoid the same frame from being created multiple times, a feedback loop must exist to enable the frame constructor to only regenerate frames that are not already in the frame cache. This helps reduce the cache pressure on the frame cache write port.
CHAPTER 4

SIMULATION ENVIRONMENT

4.1 Benchmarks

For this study, 11 of the SPEC2000 integer benchmarks are used. The benchmark *perlbench* is not used because of problems in running it within the simulation environment. All benchmarks are simulated up until completion or for a maximum of $5 \times 10^8$ instructions. Table 4.1 shows the number of simulated instructions for each benchmark.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Instructions</th>
<th>Input Set</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>bzip2</em></td>
<td>$2.89 \times 10^8$</td>
<td>modified SPEC test input</td>
</tr>
<tr>
<td><em>crafty</em></td>
<td>$5 \times 10^8$</td>
<td>modified SPEC test input</td>
</tr>
<tr>
<td><em>con</em></td>
<td>$5 \times 10^8$</td>
<td>SPEC test input (cook)</td>
</tr>
<tr>
<td><em>gap</em></td>
<td>$4.90 \times 10^8$</td>
<td>modified SPEC test input</td>
</tr>
<tr>
<td><em>gcc</em></td>
<td>$2.84 \times 10^8$</td>
<td>jump.i -o jump.o</td>
</tr>
<tr>
<td><em>gzip</em></td>
<td>$5 \times 10^8$</td>
<td>modified SPEC test input</td>
</tr>
<tr>
<td><em>mcf</em></td>
<td>$4.13 \times 10^8$</td>
<td>modified SPEC train input</td>
</tr>
<tr>
<td><em>parser</em></td>
<td>$1.25 \times 10^8$</td>
<td>modified SPEC test input</td>
</tr>
<tr>
<td><em>twolf</em></td>
<td>$5 \times 10^8$</td>
<td>modified SPEC train input</td>
</tr>
<tr>
<td><em>vortex</em></td>
<td>$2.65 \times 10^8$</td>
<td>modified SPEC train input</td>
</tr>
<tr>
<td><em>vpr</em></td>
<td>$5 \times 10^8$</td>
<td>modified SPEC test input</td>
</tr>
</tbody>
</table>

All benchmarks have been compiled using the Compaq Alpha C compiler DEC C V5.9 with optimization level 4. At this level of optimization, the compiler performs in-lining, loop unrolling, and code replication to eliminate branches.
4.2 Simulation Environment

The simulation framework is built upon the Alpha instruction-level simulator provided as the core of the SimpleScalar 3.0 tool set.

A microarchitectural timing simulator has been created to model per-instruction processing delays associated with a dynamically scheduled pipeline, including wrong path effects. The rePLay frame constructor, optimization engine, and frame cache are coupled to this microarchitectural simulator to measure the effects of different frame cache configurations.

Since the rePLay engines inject their own instructions into the original instruction stream (for example, the frame constructor converts branches into ASSERT instructions), the Alpha ISA has been augmented to include the rePLay internal ISA. The rePLay ISA contains several versions of assertion instructions corresponding to the varieties of conditional branch, indirect branch, and return instructions in the Alpha ISA, as well as instructions to move register values to/from internal rePLay registers. These move instructions have been added to facilitate dynamic optimization (although for the experiments in this thesis the optimization engine is turned off).

4.3 Baseline Processor

For the evaluation presented, an aggressive eight-wide dynamically scheduled execution engine is modeled. This processor supports fetch, decode, and execute of eight instructions per cycle.

The fetch engine of this processor consists of an 8-kbyte instruction cache that delivers up to eight instructions in one cycle up until the branch and a 128-kbyte frame cache using one of the two schemes described in Section 3. The baseline processor also includes a 256-set/8-way BTB, 14 bits of history in the gshare branch predictor, a branch bias table (BBT) with 32K entries for direct conditional branches in the frame constructor, and a BBT with 4K entries for indirect conditional branches.
The execution engine of this processor contains a 64-kbyte data cache that can support up to three loads or stores per cycle (in any combination). The memory disambiguator perfectly predicts if newly arriving memory instructions are dependent on any instructions that are in-flight. The instruction window allows up to 1024 instructions to be in-flight.

For this thesis, the rePLay optimization engine performs no optimizations. Here, the focus is on the ability of the frame caching schemes to effectively cache frames.
CHAPTER 5

PERFORMANCE ANALYSIS

In this chapter, the chaining and pooling schemes are analyzed over various design parameters in order to characterize the behavior of each. The optimal 128-kbyte configuration for each frame cache scheme is then chosen and the behavior of each more deeply studied, paying special attention to how efficiently each scheme stores frames over a program’s execution. Note that in both schemes, a cache line can hold eight 32-bit instructions. Therefore, a 128-kbyte cache (not including overhead) has 4K total lines.

5.1 Frame Cache Associativity Analysis

Since the main interest is in the effectiveness of the frame caching mechanism and not purely on the overall performance of the microarchitecture, three metrics that isolate the effects of the frame cache are primarily used: (1) the percent of the dynamic instruction stream covered by frames, (2) the average length of frames fetched, and (3) the completion rate of frames. These measurements isolate cache behavior in ways similar to how hit rate measures the effectiveness of a traditional data cache. In particular, dynamic istream coverage is a direct measurement of how effective the frame cache is at delivering useful frames to the execution core. Average frame length is a metric that indicates how large the average window of instructions within which the optimization engine could work would be. Completion rate shows how reliable the frames delivered by the frame cache are (how likely an assertion within a fetched frame is NOT to fire).
A quantitative study in the design space of the chaining scheme is first presented. The first significant factor on performance of the cache is its associativity. The nature of the chaining scheme, by which it allows any cache line to hold head or body segments of a frame, leads to some complex conflict behavior patterns. The number of sets and associativity are varied from a direct-mapped configuration to a 64-way configuration all with 4K cache lines. Figure 5.1 shows the dynamic frame coverage of the instruction stream for the various chaining configurations averaged over all the benchmarks. Included is the coverage attainable with an infinitely large frame cache.

![Graph showing average % coverage of instruction stream for various chaining configurations.](image)

**Figure 5.1:** Percentage of instruction stream covered by frames for various chaining associativities

There is a significant increase in coverage as the associativity of the cache is increased from direct-mapped to eight-way associative. The 512-set/8-way chaining frame cache is therefore chosen as the baseline. It achieves nearly the same instruction stream coverage as the more highly associative configurations while retaining a practical implementation cost. Jumping from 8-way set-associative to 16-way is an expensive hardware leap that would achieve only an incremental improvement in frame coverage of the dynamic instruction stream.

Next, various configurations for the pooling scheme are examined. For this scheme, there are three variables to consider in organizing the 4K cache lines: the number of sets and associativity of the head partition and the number of lines in the direct-mapped body partition. Figure 5.2 shows the dynamic instruction stream coverage for various configurations, again averaged over all the benchmarks. Each group of five bars in the figure shows different
associativities for a particular number of cache lines in the head partition. For example, the first group shows variations on a pooling scheme in which 128 of the 4K lines are allocated for the head partition and the rest are used in the body partition. The first bar in the group represents a 128-set/direct-mapped head partition, the second a 64-set/2-way configuration, the third is 32-set/4-way, the fourth is 16-set/8-way, and the last is 8-set/16-way. The bars in the second group follow the same pattern for a cache with 256 head lines, and so forth.

![Average % Coverage of Istream](image)

**Figure 5.2:** Percentage of istream covered by frames for various pooling associativities

In examining the data, one can see that as the number of sets increases, the associativity beyond direct-mapped becomes less of a factor in boosting performance. As the number of head blocks increases, the dynamic coverage shown in Figure 5.2 becomes less sensitive to associativity and more sensitive to the number of body cache lines. While the differences in dynamic coverage are not drastic between the various partitionings, it seems to peak in the groups with 512 and 1K head blocks. As the number of head cache lines increases, the lack of body lines begins to affect coverage. In choosing which scheme will be the pooling baseline, a 16-way head partition is ruled out for the same reasons as above for chaining. Comparing the 8-way schemes in the two peak groups shows a slight edge in coverage going to the configuration with a 128-set/8-way head partition and 3K cache lines in the body partition. Therefore, this will be the pooling baseline.

Table 5.1 shows the average length of frames fetched (Avg Len), the percentage completion of those frames (FCR), and the dynamic instruction stream coverage (Cov) for both the
chaining scheme and the pooling scheme. As can be seen, the variation in these metrics of the two schemes is minimal with the pooling scheme offering slightly better characteristics overall. Subsequent studies demonstrate how the pooling scheme, when compared on other dimensions, is a more desirable technique for building a frame cache.

Table 5.1: Average length of frames fetched, percentage completed, and percentage coverage of istream

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>512-set/8-way Chaining</th>
<th>128-set/8-way Pooling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg Len</td>
<td>FCR</td>
</tr>
<tr>
<td>bzip2</td>
<td>53</td>
<td>95.6</td>
</tr>
<tr>
<td>crafty</td>
<td>46</td>
<td>97.9</td>
</tr>
<tr>
<td>con</td>
<td>65</td>
<td>92.2</td>
</tr>
<tr>
<td>gap</td>
<td>44</td>
<td>97.9</td>
</tr>
<tr>
<td>gcc</td>
<td>28</td>
<td>98.6</td>
</tr>
<tr>
<td>gzip</td>
<td>57</td>
<td>99.1</td>
</tr>
<tr>
<td>mcf</td>
<td>27</td>
<td>98.4</td>
</tr>
<tr>
<td>parser</td>
<td>23</td>
<td>99.2</td>
</tr>
<tr>
<td>twolf</td>
<td>38</td>
<td>99.5</td>
</tr>
<tr>
<td>vortex</td>
<td>55</td>
<td>99.3</td>
</tr>
<tr>
<td>vpr</td>
<td>53</td>
<td>98.9</td>
</tr>
<tr>
<td>AVG</td>
<td>44</td>
<td>97.9</td>
</tr>
</tbody>
</table>

5.2 Frame Cache Size Analysis

Another important metric to consider in designing a frame cache is total storage size. For this study, percent frame coverage of the istream is examined for various frame cache sizes, ranging from 8 kbytes to infinitely large. Both chaining and pooling schemes are eight-way set-associative. For chaining, the number of sets is varied to change size. For pooling, both the number of sets in the head partition and the number of body partition cache lines are varied to maintain a ratio of three lines in the body partition for every line in the head partition. This is done because the analysis in the previous section indicates a near-optimal performance for this ratio in a 128-kbyte scheme. The exhaustive study required to find optimal performance as head partition size and associativity, body partition size, and
overall cache size vary stretches beyond the focus of this thesis. The two-dimensional study provided in Figure 5.3 provides a reasonable approximation of how overall cache size affects performance. As illustrated, the 128-kbyte cache size achieves coverage that is reasonably close to an infinitely large frame cache.

![Bar chart showing % Frame Coverage of Istream vs. Frame Cache Storage Size](image)

Figure 5.3: Percentage of istream covered by frames as a function of overall cache size

### 5.3 Performance Numbers for Chaining vs. Pooling

While this thesis is not so much exclusively a study of the performance of frame cache schemes as it is an examination of the complex behaviors present in the cache, execution time is still an important metric for comparison. Figure 5.4 shows the execution times for each benchmark using both chaining and pooling, along with the fraction of the total cycles spent executing frames. As illustrated, the pooling scheme achieves a shorter execution time than chaining for most benchmarks, with an average speedup of 2.5%. For bzip2 and gzip, the execution times are almost the same, and for gap and mcf, chaining actually performs slightly better. These exceptions can be explained based on a deeper study into the behaviors present in each frame cache scheme. In particular, Chapter 6 discusses how efficiently each scheme uses its total storage space. As will be discussed, there is a delicate balance in how
to partition the pooling scheme in terms of the head-to-body size ratio, and finding a ratio that satisfies all programs is very difficult. This and other concepts are examined in the following sections as chaining and pooling are compared from several angles.

![Average speedup of pooling over chaining = 2.5%](chart.png)

Figure 5.4: A comparison of total execution times for chaining vs. pooling broken down into normal (nonframe) and frame cycles

### 5.4 Frame Entry Lifetime Analysis

In understanding the behavior of the frame cache schemes, it is important to examine the lifetime of an average entry in the cache. This section investigates lifetimes for entries of various lengths in both the chaining and pooling schemes.

On pages 27-29, Figures 5.5 through 5.26 show average useful lifetimes for each benchmark run with each frame cache scheme. *Useful lifetime* is the number of cycles between the first read of a frame and the last read of that frame before it is evicted from the cache. Each bin is labeled with the minimum size N of frames counted in that bin. Each bin represents frames of eight different sizes. For the studies performed in this thesis, minimum frame size is set to three instructions and maximum size set to 256 instructions.

It is interesting to note the differences between chaining and pooling for each of the benchmarks. For instance, Figure 5.5 provides the average useful lifetimes of *gcc* on the chaining scheme, and Figure 5.6 provides the average useful lifetimes on the pooling scheme.
Figure 5.5: Chaining: gcc lifetimes

Figure 5.6: Pooling: gcc lifetimes

Figure 5.7: Chaining: mcf lifetimes

Figure 5.8: Pooling: mcf lifetimes

Figure 5.9: Chaining: eon lifetimes

Figure 5.10: Pooling: eon lifetimes

Figure 5.11: Chaining: crafty lifetimes

Figure 5.12: Pooling: crafty lifetimes
Figure 5.13: Chaining: *parser* lifetimes

Figure 5.14: Pooling: *parser* lifetimes

Figure 5.15: Chaining: *bzip2* lifetimes

Figure 5.16: Pooling: *bzip2* lifetimes

Figure 5.17: Chaining: *gap* lifetimes

Figure 5.18: Pooling: *gap* lifetimes

Figure 5.19: Chaining: *gzip* lifetimes

Figure 5.20: Pooling: *gzip* lifetimes
Figure 5.21: Chaining: *twolf* lifetimes

Figure 5.22: Pooling: *twolf* lifetimes

Figure 5.23: Chaining: *vortex* lifetimes

Figure 5.24: Pooling: *vortex* lifetimes

Figure 5.25: Chaining: *vpr* lifetimes

Figure 5.26: Pooling: *vpr* lifetimes
Notice that in Figure 5.5, shorter frames seem to be favored in the chaining scheme. This can be explained by the fact that every cache line in the chaining scheme can store either a head segment or body segment of a frame. When a new frame entry is being written into the cache, it may very likely overwrite part or all of an old entry in the cache. Thus, longer entries are more likely to be kicked out due to capacity conflicts on frame cache writes, and longer frames have shorter lifetimes in the chaining scheme. It should be noted that the average frame length for both schemes is roughly the same.

In Figure 5.6 the same study on gcc is done for the pooling scheme. Here, the distribution is less discriminating to longer frames. The fact that a slope still exists is because the total number of short frames written is much greater than the number of very long frames. In particular, the spike seen on the graphs for frames between length 0 to 7 instructions is because the high percentage of these frame written yields a much higher likelihood that they will come into the cache and kick out longer frames than vice-versa. Section 5.5 will explore the frame writes in more detail. The reason that pooling is more fair to longer frames is that the head and body partitioning essentially hides the length of frame entries from invalidations. If the freelist of body lines is empty during the write of a new frame, an entry in the head partition is randomly selected to release its body lines (and hence becomes invalid). Since all frames have equally sized head lines, longer frames are no more a target for eviction than shorter frames.

Note the difference in average useful lifetimes between chaining and pooling for gcc shown Figures 5.5 and 5.6. Frames cached using chaining have an average useful lifetime that is 64% fewer cycles in length than pooling for gcc because frames are more likely to be evicted with chaining.

Observing all the benchmarks leads one to see a variety of behavior patterns. Notice that for mcf, Figures 5.7 and 5.8 reveal a much higher average lifetime for frames in the chaining scheme than the pooling. This can be explained in part by the short average frame length for mcf (27 instructions for chaining, 26 for pooling). Such extremely short frames are favored in the chaining scheme because of the freedom offered in placing any frame in

30
any place in the cache. In pooling, very short frames can suffer because a great deal of the body partition space is left unused.

The anomalous behavior of con in Figures 5.9 and 5.10 can be explained by the fact that for this particular run of con on the chaining scheme, no frames between length 0 to 7 are formed. In the pooling run, only ten such frames are formed, and they last a very long time in the cache before being evicted. This is why the data look so skewed in Figure 5.10.

The other benchmarks are presented so that the reader can observe all of the other behavior comparisons present between the two schemes. While not every benchmark follows the same trend, it can be seen that in general, longer frames are treated more fairly in the pooling scheme. Notice the more even distributions of pooling over chaining for most of the other benchmarks. For instance, crafty in Figures 5.11 and 5.12 and parser in Figures 5.13 and 5.14 show particularly striking differences in lifetime distributions.

### 5.5 Frame Cache Writes

Figures 5.27 and 5.28 offer another illustrative comparison of the differences between the chaining and pooling schemes. The number of frames written is shown, separated into bins based on length, and the fraction of those frames that are never read. The data presented represent average behavior over all benchmarks. Careful examination of the graphs reveals that, in general, a somewhat higher fraction of frames are never read in chaining than in pooling. This behavior can again be explained by the poor conflict behavior of the chaining scheme. Frames are more likely to be evicted before they are read in the chaining scheme.

Notice the high percentage of short frames written. This is due to the fact that short frames are easier to form than long frames in the frame constructor because very few branches need to be biased. In particular, frames of length 0 to 7 most likely have only one or two assertions that were generated by highly biased branches, while frames closer to the maximum length of 256 instructions must generally wait much longer for all branches within the frame region to become promotable.
To aid in understanding the frame cache, Figure 5.29 on the next page shows the time spent modifying the cache. As illustrated, both schemes spend relatively few cycles during the total execution writing to the cache. Thus, one write port is sufficient for both schemes.

The higher percentages for bzip2 and econ can be explained by the fact that in these particular benchmarks, several important frames are being constantly evicted from the chaining scheme due to conflict patterns. Recall that the frame constructor only regenerates frames if they are not already in the frame cache. In bzip2 and econ, the constant eviction of some very useful frames during execution causes the frame constructor to regenerate these frames again and again, which causes an increase in activity on the frame cache’s write port.
5.6 Frame Cache Reads

It is also interesting to note the distribution of frame cache reads by frame length. Figures 5.30 and 5.31 show this distribution for chaining and pooling, respectively. Note the very similar shape of the distribution in each. This correlates with the average frame length data presented in Table 5.1 of Section 5.1, which show that average frame length barely changes from chaining to pooling.

![Frame Length (N to N+7)](image)

Figure 5.30: Chaining: Frame cache reads grouped by size

It should be noted, however, that while the read distributions are very similar between the two schemes, the total number of frame reads is higher on the pooling scheme for most benchmarks, which helps explain the higher coverage and slight overall speedup as compared to chaining. Table 5.2 reveals that an average of 1.2% more frames are fetched in the pooling scheme. Combined with the lower likelihood of highly useful frames being evicted in pooling
versus a greater chance in chaining, this metric helps explain the slight improvement in performance for the pooling scheme.

Table 5.2: Percentage increase in frames fetched for pooling over chaining

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Percent Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>-1.6</td>
</tr>
<tr>
<td>crafty</td>
<td>3.1</td>
</tr>
<tr>
<td>con</td>
<td>5.4</td>
</tr>
<tr>
<td>gap</td>
<td>0.3</td>
</tr>
<tr>
<td>gcc</td>
<td>-0.2</td>
</tr>
<tr>
<td>gzip</td>
<td>0.2</td>
</tr>
<tr>
<td>mcf</td>
<td>0.6</td>
</tr>
<tr>
<td>parser</td>
<td>0.5</td>
</tr>
<tr>
<td>twolf</td>
<td>2.9</td>
</tr>
<tr>
<td>vortex</td>
<td>1.1</td>
</tr>
<tr>
<td>vpr</td>
<td>0.5</td>
</tr>
<tr>
<td>AVG</td>
<td>1.2</td>
</tr>
</tbody>
</table>

At the same time, these very similar read distributions help explain why the pooling scheme does not provide more of a speedup over chaining than is observed. The fact that most of the frames fetched are of a shorter size caters to the chaining scheme’s strength of having freedom to use all its space for either head or body segments. In addition, the shorter frames expose the weakness of the pooling scheme in its being susceptible to wasted body
partition space, as will be discussed in Chapter 6. Though pooling is ultimately better than chaining, it is not a runaway winner in the performance race, and these data help explain why.

5.7 Wait Queue Sensitivity Analysis

As mentioned in Section 3.3.2, another implementation detail to consider is the queue where pending frames wait to be written into the frame cache. One major part of the wait queue to evaluate here is its length. Figure 5.32 illustrates the effect on performance for both chaining and pooling as the wait queue is given lengths of 10, 10³, and 10⁶ entries. An infinite wait queue is also modeled. The data reveal that the length of the queue is of little consequence for overall performance. Thus, a wait queue length of 10 entries is used.

![Figure 5.32: Effect of varying wait queue lengths on frame coverage of istream](image)

The other important factor to consider in the wait queue design is the policy used to handle the overflow condition that can occur when frames come into the queue faster than they leave to be written into the cache. As mentioned in Section 3.3.2, several options are available. Figure 5.33 examines three policies: (1) drop pending frames at the front of the queue to make room for incoming frames; (2) drop incoming frames if no room is available; and, (3) drop any frame trying to access locked space in the cache. Again, this parameter shows little bearing on performance of the cache. Therefore, dropping frames at the front of the queue is chosen as the policy because of its implementation simplicity.
Figure 5.33: Effect of wait queue drop policy on frame coverage of istream

5.8 Replacement Policies

Section 3.3.3 mentions another implementation detail to analyze: the frame cache replacement policy. Several replacement policies can be used, including least-recently-used (LRU), not-most-recently-used (nMRU), and random with preference for empty cache lines. Figure 5.34 shows the effect of the various schemes on execution time. In the figure, $sRbR$ stands for random selection for both sets and ways, $sRbL$ stands for random selection selection of sets and LRU selection of ways, $sLbL$ stands for LRU selection of sets and ways, and $sNbN$ stands for not-most-recently-used (NMHU) selection of both sets and ways. Note that a random policy with a preference for empty cache lines is just as effective as using LRU or NMHU. This random policy is easy to implement in hardware and is the one used in all other experiments.

Figure 5.34: Effect of cache line replacement schemes on frame coverage of istream
CHAPTER 6

FRAME CACHE UTILIZATION

Another important metric to consider is the amount of frame cache space that is actually in use at any given time. One may assume that the percentage of space used in the frame cache is rather high, perhaps near 100%. This assumption is based on the notion that even after frames lose their usefulness, they still must take up space in the cache and, thus, the amount of empty space should be minimal. As is evident in the following sections, however, this expected behavior is not the case. In fact, for some benchmarks, the average percent utilization of the frame cache space is quite low. Upon first glance, these low numbers seem troubling. The expectation that an efficient frame cache scheme should use most of its space is not met here. Even with the complicated conflict patterns that exist, it would seem that a good cache should have very few empty lines in the steady state execution of a program. However, as is observed and explained, the utilization of frame cache space is not so simple. There are many significant factors that not only explain why the utilization of the cache space is lower than expected but also indicate that this low utilization is not a shortcoming of the frame cache. Rather, it is simply a result of the many complexities within a frame processor.

6.1 Chaining Scheme Utilization Analysis

Figure 6.1 shows how the average percentage of space utilized in the chaining scheme varies with associativity. For chaining, cache utilization increases as associativity increases. This
illustrates the great vulnerability to write conflicts that exists in the chaining scheme. The lower the associativity, the more likely a particular frame entry is to be invalidated by an incoming frame.

![Average percent utilization of the cache space for chaining](image)

Figure 6.1: Average percent utilization of the cache space for chaining

Write conflicts alone do not explain the great lack of utilization, however. There are other key factors at work here. The two biggest contributors are frame evictions (as described in Section 3.3.4) and the high percentage of dead frames (those that are never read), as shown in Figure 5.27 of Section 5.5.

Evictions play a key role in the following way. As frames finish their usefulness and are evicted from the frame cache by the execution core, their space is not always refilled because the program moves to some new phase in which a different portion of the frame cache is exercised. In other words, write/evict activity in one phase of a program may target particular sets in the cache due to localized contexts (the hash key used to access the cache) in that phase. The program’s phases wind up taking a sort of random walk through the cache, utilizing one portion for one phase followed by evicting that phase’s frames and moving to some new portion of the cache.

Combined with the high numbers of evictions that occur to empty frame cache space, there is a more subtle behavior that can contribute to the maintenance of unused space in the frame cache. The high number of dead frames in the chaining scheme can work in concert with the already empty space caused by evictions to yield even lower utilization.
When active frames are evicted from the cache due to a write conflict, they are likely to come back into the cache because of the probability that they are still active and thus get reformed by the frame constructor. When dead frames are evicted due to write conflicts, they are not likely to come back around because they were not useful to begin with. Thus, when new frames are written into the cache, a scenario can exist in which they are written partially into already-empty cache lines and partially into dead frame space. This causes the dead frame overwritten to be invalidated, and any space unused by the new entry becomes empty. In this manner, empty space and dead frame space can simply trade cache lines back and forth, which in essence helps maintain a certain amount of empty space in the cache.

### 6.2 Pooling Scheme Utilization Analysis

The pooling scheme enjoys a better utilization of its cache space than the chaining scheme. However, there is still suboptimal utilization, as indicated in Figure 6.2. First, one can notice that the utilization does not continually increase with higher associativity. It peaks at around 70.5% for an eight-way set-associative head partition. This indicates that there are other configuration parameters at work.

![Avg % Utilization of Cache Space](image)

**Figure 6.2:** Average percent utilization of the cache space for pooling

In particular, Table 6.1 shows the breakdown of utilization for the head and body partitions of the pooling frame cache. In examining the pooling scheme, it appears that in most
benchmarks, it is body-block-limited. In other words, the average length of frames is long enough that almost all of the body partition cache lines are used up before the total number of head partition lines can be filled. In particular, notice that \textit{crafty}, \textit{con}, \textit{gap}, \textit{parser}, \textit{twolf}, \textit{vortex}, and \textit{vpr} are body-block-limited. The only benchmark that seems to use most of the cache space is \textit{gcc}.

Table 6.1: Breakdown of space utilization for a 128-set/8-way/3K-body pooling cache

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>% Head</th>
<th>% Body</th>
<th>Limitation</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>23.3</td>
<td>57.2</td>
<td>working set</td>
</tr>
<tr>
<td>crafty</td>
<td>59.5</td>
<td>98.4</td>
<td>body limited</td>
</tr>
<tr>
<td>con</td>
<td>20.4</td>
<td>98.7</td>
<td>body limited</td>
</tr>
<tr>
<td>gap</td>
<td>50.9</td>
<td>89.2</td>
<td>body limited</td>
</tr>
<tr>
<td>gcc</td>
<td>91.8</td>
<td>95.4</td>
<td>even</td>
</tr>
<tr>
<td>gzip</td>
<td>24.9</td>
<td>47.8</td>
<td>working set</td>
</tr>
<tr>
<td>mcf</td>
<td>27.1</td>
<td>39.4</td>
<td>working set</td>
</tr>
<tr>
<td>parser</td>
<td>58.1</td>
<td>84.9</td>
<td>body limited</td>
</tr>
<tr>
<td>twolf</td>
<td>54.8</td>
<td>98.1</td>
<td>body limited</td>
</tr>
<tr>
<td>vortex</td>
<td>44.3</td>
<td>97.7</td>
<td>body limited</td>
</tr>
<tr>
<td>vpr</td>
<td>39.1</td>
<td>97.1</td>
<td>body limited</td>
</tr>
</tbody>
</table>

The other benchmarks, \textit{bzip2}, \textit{gzip}, and \textit{mcf}, do not use up enough cache space to become limited in either the head or body partition. This can be explained by the very small working sets of frames for these benchmarks. This means that the total number of unique frames written for these benchmarks is much lower than for the other benchmarks. Thus, most of the benchmarks target all areas of the cache pretty evenly while these three continually target specific regions (meaning they have many write conflicts).

Table 6.2 shows the same breakdown for a pooling cache with a 64-set/8-way head partition and a body partition with 3584 lines. Notice that many of the benchmarks that were previously body-block-limited have become head-block-limited, in particular \textit{gap}, \textit{parser}, and \textit{twolf}. The ratio of head lines to body lines in the cache has become 1:7, while in the scheme above it is 1:3. For the head-limited benchmarks below, there is not enough head partition space to satisfy the number of entries written to the cache. Given the short average frame
length of most of the benchmarks, it makes sense that with a large set of small frames, much of the body partition space can go unused.

Table 6.2: Breakdown of space utilization for a 64-set/8-way/3584-body pooling cache

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>% Head</th>
<th>% Body</th>
<th>Limitation</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>43.0</td>
<td>46.2</td>
<td>working set</td>
</tr>
<tr>
<td>crafty</td>
<td>94.9</td>
<td>73.7</td>
<td>head limited</td>
</tr>
<tr>
<td>eon</td>
<td>46.2</td>
<td>98.4</td>
<td>body limited</td>
</tr>
<tr>
<td>gap</td>
<td>88.1</td>
<td>71.1</td>
<td>head limited</td>
</tr>
<tr>
<td>gcc</td>
<td>98.4</td>
<td>46.4</td>
<td>head limited</td>
</tr>
<tr>
<td>gzip</td>
<td>44.9</td>
<td>36.6</td>
<td>working set</td>
</tr>
<tr>
<td>mcf</td>
<td>47.1</td>
<td>29.7</td>
<td>working set</td>
</tr>
<tr>
<td>parser</td>
<td>82.2</td>
<td>54.1</td>
<td>head limited</td>
</tr>
<tr>
<td>twolf</td>
<td>94.7</td>
<td>78.7</td>
<td>head limited</td>
</tr>
<tr>
<td>vortex</td>
<td>92.0</td>
<td>93.3</td>
<td>even</td>
</tr>
<tr>
<td>vpr</td>
<td>82.2</td>
<td>89.1</td>
<td>even</td>
</tr>
</tbody>
</table>

Also notice that vortex and vpr have become somewhat evened out in the second configuration. For these benchmarks, the 1:7 partitioning is near optimal. Also note that eon is still body-limited. This benchmark would prefer an even smaller number of head lines in favor of more body lines.

From Tables 6.1 and 6.2, one can see that a complication in designing the pooling scheme is in finding the proper head-to-body partitioning. In many cases, this ratio can be application specific. This makes the chaining scheme’s freedom to use all cache lines as either head or body segments more appealing. Overall, however, the higher number of write conflicts in the chaining scheme makes the pooling scheme a still more effective design of the frame cache.

6.3 Further Comparison of Chaining and Pooling

For the edification of the reader, average occupancy per set is discussed for each scheme. For chaining, all cache space is represented; this includes sets 0 through 511. For pooling,
only the head partition is represented as it is the only set-associative portion of the cache; this includes sets 0 through 127. Each scheme has eight cache lines per set, and a bar on the graph represents the average number of those lines that are occupied with a valid entry.

As an example, Figure 6.3 shows the average set occupancy of con in the chaining scheme. Its average set occupancy appears to be around five cache lines.

![Figure 6.3: Chaining: con set utilization](image)

Figure 6.3: Chaining: con set utilization

Figure 6.4 shows the pooling scheme set utilization for con. Note that the head partition does not appear to be well utilized while the cache space in the chaining scheme above appears more evenly distributed. This can again be explained by the body-partition limitation as discussed in the previous section. In the chaining scheme, utilization looks more even for con because of the freedom to use cache lines for either head or body segments. A similar discussion can be applied to many other benchmarks, as well.

![Figure 6.4: Pooling: con set utilization](image)

Figure 6.4: Pooling: con set utilization

Figures 6.5 through 6.24 on the next few pages show the same comparison for all other benchmarks. Notice the even utilization for some benchmarks and the uneven utilization for others. The uneven utilization can again be attributed to small working sets of frames along with localized write conflict behavior.
Figure 6.5: Chaining: *bzip2* set utilization

Figure 6.6: Pooling: *bzip2* set utilization

Figure 6.7: Chaining: *crafty* set utilization

Figure 6.8: Pooling: *crafty* set utilization

Figure 6.9: Chaining: *gap* set utilization

Figure 6.10: Pooling: *gap* set utilization

Figure 6.11: Chaining: *gcc* set utilization

Figure 6.12: Pooling: *gcc* set utilization
Figure 6.19: Chaining: *twolf* set utilization

Figure 6.20: Pooling: *twolf* set utilization

Figure 6.21: Chaining: *vortex* set utilization

Figure 6.22: Pooling: *vortex* set utilization

Figure 6.23: Chaining: *vpr* set utilization

Figure 6.24: Pooling: *vpr* set utilization
CHAPTER 7

A GLANCE AT REDUNDANCY

Because of the dynamic nature under which frames are constructed, the potential for redundantly caching a single instruction in the frame cache exists. As a result, the frame cache space will be used less effectively than if the redundancy did not exist. Similar effects have been observed on trace cache systems, and schemes to limit the effects of this redundancy have been proposed. The block-based trace cache was proposed as a scheme to limit the amount of redundancy [13], and Red and Blue Traces were developed to limit the redundancy between the trace cache and the adjoining instruction cache [7].

Here, a measurement of frame cache redundancy is provided. Figure 7.1 shows a distribution of the average basic block redundancy in both the chaining and pooling schemes. These data are collected by averaging the block-level redundancy on the pooling scheme. Furthermore, it is a time-average for each benchmark. During execution, the frame cache is examined once every $10^4$ cycles in order to take a snapshot of redundancy. The data shown represent a time-average across all these snapshots.

Each bin on the x-axis represents the number of times a basic block address is repeated in the frame cache at any given time, and the magnitude of a bar represents the average number of block addresses for a given repetition count. In particular, the first bar represents all non-redundant basic block addresses, the second bar represents all basic block addresses repeated twice in the frame cache for a particular instant of the execution, the third bar represents three-time redundancy, and so on. The last bin is a catch-all representing 20 or more times
repeated in the frame cache. As an example, the pooling scheme has 88 addresses repeated twice in the cache, which yield 176 total addresses for the “2” bar on the x-axis. Note that the magnitude difference between chaining and pooling stems from the fact that the pooling scheme has a higher average utilization than the chaining scheme. Hence, more basic blocks exist in the pooling scheme at any given time. Despite this magnitude difference, the general shape of the redundancy distribution curve is the same for both schemes. The distribution shows that redundancy is severe, with many blocks having multiple copies. For example, in the pooling scheme, approximately 262 blocks have only one copy, 88 blocks have two copies (thus, 176 total basic block addresses), three blocks have 44 copies (132 total addresses), and so on. Furthermore, taking the average number of instances of a particular basic block in the cache at a given time yields 2.6 instances for chaining and 2.9 for pooling.

Exploring possible solutions to reducing redundancy is beyond the scope of this thesis. Future work to be done might include studying the concepts involved in dependency tree compression by the exploitation of dataflow redundancy. This is a higher level of redundancy compression than the block-level redundancy presented here.
CHAPTER 8

CONCLUSION

This thesis presents design alternatives for implementing a cache to store instruction entities spanning multiple basic blocks. In particular, it focuses on the design of such a cache for storing long, atomic regions called frames. Two main schemes are presented, one called chaining and another called pooling. The chaining scheme for the frame cache is a single, highly associative cache able to store frames over multiple cache lines by chaining the segments together with forward pointers in a circularly linked fashion. The pooling scheme is composed of two structures. The first is a highly associative cache for storing head segments of frames, and the second is a direct-mapped structure used for storing the body segments of frames.

While both schemes have positive and negative attributes, the pooling scheme outperforms the chaining scheme for four important metrics: coverage of the dynamic instruction stream by frames, usage of the frame cache’s storage space, fairness of eviction for frames of various lengths, and total execution time. The overall dynamic coverage of the 128-set/8-way/3K-body pooling scheme is 81.7%, versus 79.7% for the 512-set/8-way chaining scheme. During execution, an average of 70.5% of the storage space in the pooling cache contains valid entries, versus only 51.0% for chaining. As the distribution of lifetime in the frame cache versus length in Figures 5.5 through 5.26 illustrates in Section 5.4, the pooling scheme is more fair to long frames than the chaining scheme. Chaining, because of the nature of its single structure, tends to evict longer frames over shorter ones in random replacement due
to a write conflict. Finally, the pooling scheme shows an average speedup of 2.5% over the chaining scheme across all the benchmarks. Certainly, chaining offers more freedom than pooling in terms of how the cache lines are used, but the overall performance of the pooling is better. Thus, the pooling scheme is the recommended method for building a frame cache.

This thesis also presents some deeper behavior analysis into the activity of an average frame entry over its lifetime in the cache. Several lifetime statistics are shared and explained in order to give the reader a clear sense of how the frame cache behaves within its frame processor system. Specific implementation details such as the write port, locking mechanism, and wait queue are discussed to reveal the complexities of frame cache design. A study of how efficiently space is used in each scheme is also presented. Finally, a brief glimpse into the potential redundancy of information in the frame cache is introduced.

The frame cache is a complex entity with a large design space. This thesis aims to pave the way for further developments in designing storage facilities for long regions of instructions in the ever-growing arena of dynamic instruction region reuse. The concepts presented here can be applied to trace cache design and other forms of long instruction region storage, as well. The main contribution of this thesis is in providing an understanding of the many design alternatives that come with this complex structure.
REFERENCES


