Software Based Instruction Caching for the RAW Architecture

by

Jason Eric Miller

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 21, 1999

Author . . . .
Department of Electrical Engineering and Computer Science
May 20, 1999

Certified by . . . .
Anant Agarwal
Professor of Computer Science
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Software Based Instruction Caching for the RAW Architecture

by

Jason Eric Miller

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 1999, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis addresses the design and implementation of a software based instruction caching system for the RAW architecture. This system is necessary to allow large programs to be run in the limited on-chip memory available for each RAW tile. Similar systems were examined and various design issues were examined in detail. A partial system was implemented in the RAW compiler in order to gauge the feasibility of such a system. Performance data was collected from various benchmarks. The implications of this data and directions for further research are discussed.

Thesis Supervisor: Anant Agarwal
Title: Professor of Computer Science

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The RAW architecture is an example of an exposed, parallel architecture. The details of the multiple execution units and their interconnection are exposed to the compiler so that it can manage resources efficiently. Complex features such as virtual memory, out-of-order execution and caching are implemented in software to allow for customization on a program by program basis.

The remainder of this chapter describes the RAW architecture in more detail and outlines the basic issues involved in implementing a software based caching system.

Chapter two examines the major design variables in more detail and discusses the advantages and disadvantages of several options for each.

Chapter three describes the system which was implemented and reports on the results of adding software based instruction caching to a program.

Chapter four discusses possible future work and provides some concluding comments.

1.1 The RAW Architecture

The RAW architecture is based on two main ideas: providing many resources to be used in parallel and exposing the details of the architecture to allow flexibility in the way these resources are used [1, 9]. A RAW processor consists of many small, replicated computational tiles, each with it's own instruction stream. Each tile is composed of a simple pipelined RISC core (with a MIPS instruction set [6]), separate data and instruction memories and a programmable switch to communicate with other tiles. Many of the complicated features found in modern microprocessors are not implemented in hardware in a RAW system. This is done to allow the compiler to implement and customize these features as needed for a specific program.

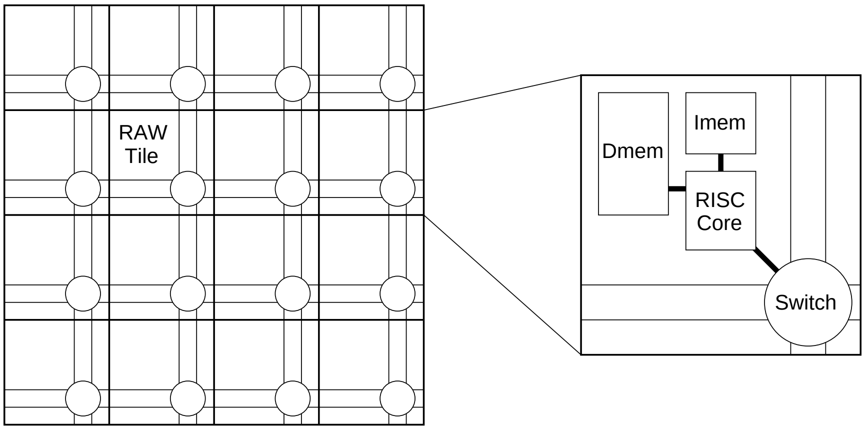


Figure 1-1: Diagram showing interconnection of RAW tiles and composition of each tile.

The tiles are interconnected in a 2D mesh (see Figure 1-1), allowing each tile to communicate directly with the four tiles adjacent to it. Communication can either occur statically by producing instruction streams for the switches at compile time, or dynamically by sending data over a separate network with a destination tile address. Dynamic messages are routed to their destinations by the switches between the two nodes. Data is only able to move to an adjacent switch on each clock cycle so communication with distant tiles requires extra clock cycles.

In order to allow for the maximum number of tiles on a processor, the data and instruction memories are kept small. Typical sizes might be in the 16 to 32 kbyte range. Since many applications are likely to require more storage than this, some mechanism is needed to load these memories with new data from a larger external memory. In effect, the tile memories will be used as caches. In keeping with the RAW philosophy, this caching behavior should be added to a program by the compiler.

1.2 Caching Overview

Due to the differences in the way data and instructions memories are used, it makes sense to devise separate, custom strategies for caching each. This thesis addresses the problem of implementing a caching system for instruction memories.

Code execution is, by its very nature, dynamic. Loops and conditionals make it possible for some code to be executed thousands of times while other code is never executed at all in a way that is impossible to predict at compile time. A compiler can only determine where flow of control *can* pass, not where it *will* pass. The same program can behave very differently on different sets of input data. Therefore, instruction caching must be implemented as a runtime system. Loading of new code cannot be scheduled at compile time but must instead be performed on demand.

Implementing caching in software as opposed to hardware has both advantages and disadvantages. The most obvious disadvantage is that the comparisons and lookups needed to determine if a certain piece of data is in the cache require multiple processor instructions in a software managed cache but can be performed in parallel in a hardware cache. On the other hand, a software implementation has an advantage in being able to change the caching scheme at will. It also allows the compiler to get involved. Thus, the compiler can use its knowledge of a program to customize the caching for that program. For example, if the compiler can determine that the entire program will fit within the instruction memory, it can remove the software caching all together.

1.2.1 Similar Systems

Software caches must do things like searches and comparisons in a serial manner instead of the parallel manner used in hardware caches. For example, a fully associative hardware cache can compare the tags of all the cache lines to the desired address simultaneously [10, p. 482] while a software based cache must check each tag sequentially. Therefore, it is not very practical to simply imitate a hardware cache in software. Instead, we should look to other software systems for inspiration. Simulation/profiling and dynamic code generation systems both face situations similar to instruction caching.

Simulation programs have the task of imitating a certain computer system by translating machine code into the machine code of the host computer. These types of programs also tend to include support for inserting extra code to collect statistics (profiling) or provide debugging information. These systems typically do their translation at runtime so that the data to be collected can be altered on the fly [11, 3] and so that time is not wasted translating code that does not actually get executed.

Dynamic code generation systems [5] operate in a very similar manner. They typically implement some virtual machine by translating an intermediate language (*e.g.*, Java bytecode) into machine code at runtime. This is done so that the same program can be run on any system which implements the appropriate virtual machine, without requiring a separate translation pass. Since the whole point of this type of system is avoiding an extra compilation pass, the translation must be done at runtime. To avoid performing this expensive translation on code which is never executed, these systems usually do translation on demand.

Both of these types of systems are frequently implemented using a *translation cache* [3, 11].

As code is translated, it is placed into the cache to be reused if that code is executed again. The translation cache is, in essence, an instruction cache. In an instruction cache, code is loaded from main memory into the cache memory and kept there as long as possible in case it is needed in the future.

However, there are several important differences between these systems and a software based instruction cache. First, the translation cache is typically stored in the main memory of the host computer and therefore can be very large [2]. A translation cache of 4 or 8 MB is large enough to hold all of the code for many programs. With today's workstations frequently having between 128 and 512 MB of RAM, there is no reason why the translation cache could not be even larger than 8 MB so that it could accommodate all but the very largest programs. Because of this, simulation and dynamic code generation systems only need to deal with their caches becoming full on an infrequent basis. This allows them to use expensive but simple methods to cope when it does occur. However, with an instruction memory of only 16 kbytes, our cache will constantly be full and space will need to be cleared for each new piece of code to be loaded. This means that deallocation needs to be just as fast and efficient as allocation.

The second big difference is that an instruction cache only loads the code into memory instead of translating it. Since the simulation programs spend most of their overhead time doing the translation or collecting statistics, the mechanism used to implement the caching can be more complex without incurring a noticeable penalty. Instruction caching code must be very efficient in order to maintain performance comparable to a system which does not need caching.

1.2.2 Basic Operation

At the most basic level, an instruction cache divides an instruction stream into some type of pieces (called *blocks*) and then loads those blocks from a large, distant (slow) memory to a small, close (fast) memory when they are needed. Hopefully, these blocks will be needed more than once, allowing them to be retrieved from the small, close memory after the first use. In a hardware cache, the blocks chosen are usually a contiguous block of two to four instructions chosen based on their alignment in memory. In a software based cache, there is an opportunity to use the compiler to create a more intelligent scheme.

In the RAW system, the software caching code will be integrated into the program by the compiler. The compiler will add a piece of code (called the *dispatch* code) which checks to see if a certain piece of code is resident in the cache, loads it if it is not and then transfers control to that code. This is very similar to the tag checks and data fetch performed by a hardware cache. However, rather than performing this check for every instruction that is executed (as would be done in hardware) the compiler will only insert jumps to our dispatch code at points where it thinks the upcoming code might not be resident in the cache. In places where it knows the code will be resident, the program can simply continue without the check.

The dispatch code will load an entire block into the cache at a time. As long as the processor is executing code within this block, the program does not need to check to see if the next code is present. The program will only need to jump to the dispatch code when execution moves to a different block. Although it initially seems like it might be hard to detect when the program enters or exits a block, the compiler can leverage its knowledge of control flow to greatly simplify this.

Chapter 2

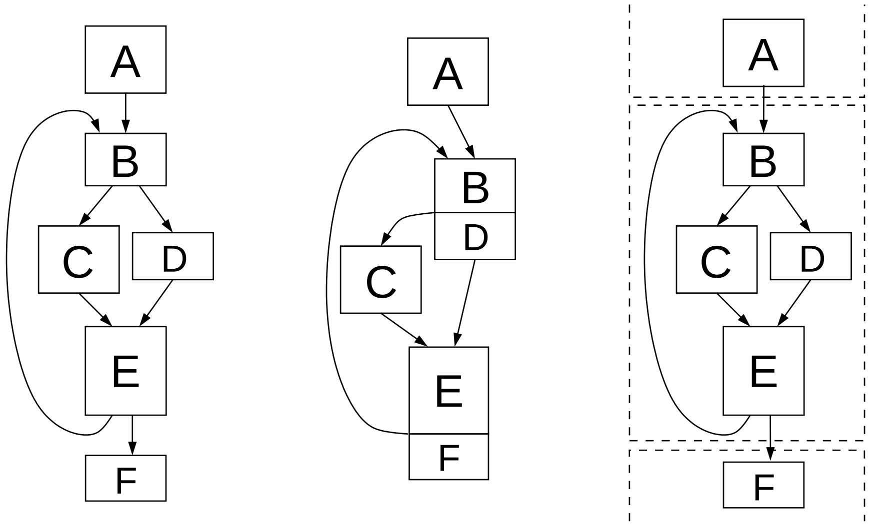
Major Design Issues

In order to implement the basic structure outlined above, several mechanisms are needed. It must be possible to find out if a block is in the cache. This should be fast since it will need to be done frequently. If the block that is needed is already present in the cache, the dispatch code will simply transfer control to it. Otherwise, it will need to be able to load a block from memory into the cache. This can be slower since it should occur less frequently. Finally, a block may need to be removed from the cache in order to make room for the block which is being loading. Each of these mechanisms will be affected by several design choices including how blocks of code are formed, how instruction memory is organized (*e.g.*, heap vs. direct mapped cache), whether blocks can move once they have been loaded and whether or not blocks of code can be modified for optimization purposes.

2.1 Block Size

The way in which a program is broken up into blocks has a very large effect on all the components of the system. Ideally, blocks should be large so that jumps to the dispatch code would be infrequent. It might also be good to have a fixed size for blocks so that it is not necessary to keep track of the size of each block and so that organization of the instruction memory can be simplified. However, breaking a program into arbitrary blocks can create more problems. For example, if a block has more than one entry point[1], then it will be harder to keep track of which entry points are in the cache. Also, loading a large block of code may waste space in the cache by loading code that will

[1] An *entry point* is a place that the dispatch code might be required to transfer control to, *i.e.*, a branch destination.



(a) (b) (c)

Figure 2-1: Control flow graphs demonstrating (a) basic blocks, (b) basic blocks joined into extended basic blocks and (c) clusters of basic blocks (shown with dashed boxes).

be skipped over by a conditional branch.

2.1.1 Basic Block

Fortunately, compilers already break programs up into blocks which have a lot of good properties. A *basic block* is a sequence of instructions whose only entry point is the first instruction and whose only exit point is the last instruction. Once control enters a basic block every instruction within the block is executed and control flow within the block is sequential. This is the block size which was chosen for our system.

Using a basic block for a cache block has several benefits. First, all instructions which are loaded will be executed, thus wasting the minimum amount of space. Second, keeping track of entry points is equivalent to keeping track of blocks, thus simplifying bookkeeping. Third, because basic blocks end whenever a change in flow can occur, every branch instruction will end a block and every label will start a new one. Therefore, making the program jump to the dispatch code when leaving a block is as simple as replacing all the branches with jumps to the dispatch code and inserting jumps before each label.

However, there are also some negative aspects to using basic blocks for a cache block. First, basic blocks are fairly small. Most programs have average basic block sizes around 6 or 7 instructions [11]. This means that jumps to the dispatch code will be frequent and overhead will be high. In addition to being small, basic blocks are also highly variable in size. Some may be only a single instruction while others may be dozens of instructions. This makes arranging them in memory more complicated. If the blocks are placed at fixed locations in memory (*e.g.*, a hash of the starting address of the block) then space may be wasted between blocks if they are too short or blocks may overlap if they are too long. If blocks are packed densely into memory, then it is much harder to deallocate them because the holes that are created may not fit the new block that needs to be placed there.

2.1.2 Extended Basic Block

One alternative to using basic blocks is using *extended* basic blocks. Extended basic blocks have a single entry point but might have multiple exit points. In terms of cache behavior, they would be very similar to basic blocks except that they would tend to be bigger. This could potentially reduce the time spent transferring data from external memory because larger, more efficient "burst" reads could be used. However, this also increases the likelihood of loading code which will not actually be executed because the branches in the middle of the extended basic block might always skip the code at the end of it. Branches which occur in the middle of a block would need to be modified so that the "taken" case would jump to the dispatch code but the "fall-through" case would simply fall-through. Branches at the end of blocks would need to be modified as with basic blocks. Using extended basic blocks will reduce the number of calls to the dispatch code but only slightly.

2.1.3 Clusters of Basic Blocks

Using groups of basic blocks with internal branches might produce larger blocks and also reduce the number of jumps to the dispatch code. The clusters would have a single entry point and one (or maybe many) exit points but may also have internal branches which have destinations within the cluster. These internal branches could be left as-is while the branches with destinations outside the cluster would be translated into jumps to the dispatch code. Clusters were inspired by the *macro-instructions* of [8]. As long as the cluster has only one entry point, bookkeeping will be just as easy as it would be using basic blocks. However, code may be loaded that is never executed and a more complex compiler will be required to find these clusters and change only the appropriate branches.

2.1.4 Fixed Size Blocks

In order to avoid the complexity and/or wastefulness of placing variable sized blocks into the cache memory, a system could be designed using fixed size blocks. The code could be divided into fixed size segments while still maintaining proper flow of control by inserting a jump instruction at the end of each segment which jumps to the beginning of the next segment. Since there can now be multiple entry points within a block, it will be harder to determine whether or not a specific entry point is in the cache when it is needed. Rather than simply assuming that there is only one entry point in a cache block and it is at the beginning, a method for finding out which entry points are within a cache block will be required. Also, this method of forming blocks may load quite a bit of code into the cache that is never executed. Finally, the insertion of all the extra jumps will adversely impact performance.

2.2 Instruction Memory Organization

Instruction memory organization is just as important to cache performance as block size. A balance must be found between space utilization efficiency and speed of allocation and deallocation. Space efficiency will influence the cache hit rate since wasted space could have been used to hold useful code. However, a more complex scheme which is able to more efficiently manage memory may take longer to place new blocks and find old blocks to throw out. For simplicity and flexibilty reasons, our system uses a heap organization as described below.

2.2.1 Associative Cache

Traditional hardware caches generally use fixed size blocks and some degree of associativity to determine where to put those blocks. The cache is conceptually a collection of numbered slots. The physical starting address of a block is hashed somehow to produce a number which corresponds to a specific slot. In a direct-mapped cache (equivalent to 1-way associativity) each slot can hold one block. In a 2-way associative cache, each slot can hold two blocks to avoid thrashing in the case that two blocks which map to the same slot are used alternately. This can be extended to any degree of associativity desired.

This technique has the advantage of it being relatively fast to determine where a block should be placed in the cache. It is also easy to determine which blocks should be thrown out because it is simply whatever block is already in the slot. However, it has the potential to be very inefficient in its memory usage. First, a block may be thrown out because of an overlap in hash values even though there is plenty of space somewhere else in the cache for that code. Second, it can be difficult to reconcile a variable block size with this scheme. Since the slots are fixed size, space will be wasted if a block is loaded that is not as big as the slot and large blocks will need to be broken up so that they will fit into the slots.

2.2.2 Heap

The opposite extreme to the highly structured associative cache model is the unstructured heap. Blocks are simply loaded into the heap starting at the first available address and using as much space as they need. The result will be that all of the blocks will be densely packed into memory. If basic blocks are used as the cache blocks, this will result in perfect efficiency of memory usage. There will be no wasted space and no unnecessary code will be loaded. Instead of using a hash to place blocks, the empty space in the heap must be kept track of. However, this scheme can become very complex when deallocation of blocks is allowed.

If the system allows *any* block to be deallocated, it will become very hard to keep track of the free space. Memory will become fragmented as blocks are loaded that do not perfectly fit the space that was cleared for them. A system which operates this way will probably need to compact the heap periodically to remove the wasted space. This is generally a very expensive operation and will severely impact performance.

Alternatively, the instruction memory could be treated more like a stack or a FIFO, *i.e.*, allocation and deallocation can only occur at the ends of the already loaded blocks. A *stack* implementation would require blocks to be allocated and deallocated from the same end. This has the undesirable property that the most recently allocated blocks are the only ones allowed to be deallocated, thus preventing us from reusing those newly allocated blocks. A *FIFO* implementation would be better. Blocks would be allocated from one end and deallocated from the other end and the instruction memory would be treated in a circular manner. This means that the oldest blocks would be deallocated to make room for the new ones. Since the *oldest* block is an approximation to the *least recently used* block, this is a much better replacement strategy. While a FIFO replacement strategy is frequently considered mediocre for random access data caches [7], it should perform better in an instruction cache where the access pattern is mostly sequential. Also, the benefit of being able to dense pack variable sized blocks may outweigh the disadvantage of a slightly increased miss rate.

Most of the simulation and dynamic code generation systems use a heap but have a greatly simplified deallocation scheme. When the cache fills up, they flush the entire thing and start over. This is a good strategy if your cache is large and fills infrequently but is not likely to produce acceptable performance for a small cache.

2.2.3 Segmented Heap

One possible compromise between the associative and heap structures would be a *segmented heap*. In essence, the cache is an associative cache with a small number of slots where each slot is a heap rather than holding a fixed number of blocks. The associativity of this system would be variable. A hash would be used to assign each block to a slot and then empty space would be found within that slot for the block. Ideally, blocks would be packed densely. In this system, it may be practical to simply flush all the code within a slot when it fills up. Since a slot is only a small portion of the cache, only a small portion of the previously loaded code would be thrown out.

Alternatively, blocks could be assigned to slots in a different way. Initially, all blocks would be placed in the first slot. When it becomes full, blocks would be placed in the second slot and so on, until the entire cache is full. When this occurs, all of the code would be flushed from the first slot and it would begin refilling. This has the advantage that it is always the oldest code which is being flushed rather that just the code which happened to collide with the current block. Again, this FIFO replacement strategy is likely to perform well in an instruction cache.

2.3 Data Structures

Any practical system must also consider the speed and footprint of the data structures necessary to maintain the cache. The most obvious data structure to use would be a list or table which contains an entry for every block which is present in the cache. Each entry would contain the *virtual* address (the address in the external memory) and the *physical* address (the address in the instruction memory) of the block. Unfortunately, this is a horribly slow data structure since it will require a full search of the table to find out if a cache block is present. Since this is the operation which most needs to be fast, this is not an acceptable solution.

2.3.1 Array

To avoid doing a full search on the table, the entries for each block could be stored in an array which is indexed by the virtual address of the block. Finding the entry for a block no longer requires a search. However, this array will be very sparse since only the entry points of each block need to be recorded. The array indices corresponding to instructions which are not entry points would be empty. To eliminate the sparsity, a new virtual address space could be created where each block is assigned a number and this number is used to request blocks. Now there is one entry per block in the program. This is fast but requires that a table be kept in memory whose size is proportional to the total number of blocks in a program, not the number of blocks currently in the cache. Since programs could conceivable grow very large, this approach is not scalable because the table would consume all of a tile's memory. In addition, the array could still become sparse because the entries for blocks which are not currently in the cache would be empty. However, this may still be a very good solution for programs that are too big to fit in the instruction memory but are not huge. Here is where the flexibility of a software based scheme pays off. The compiler could use this fast method for fairly small programs but use a slower, more scalable method for very large programs.

2.3.2 Hash Table

Another method for eliminating the sparsity of an array is to convert the array into a hash table. Instead of indexing the array by the virtual address of the block, it is indexed by some hash of the virtual address. This provides a lookup that is almost as fast as a standard array but the hash function is chosen so that the table is a fixed size and is reasonably dense.

The problem with a hash table is that it is possible for two virtual addresses to hash to the same value. This can be minimized by picking a good hash function and a large enough table size but it can still happen and must be planned for. A common method for dealing with collisions is *chaining* [4, p. 223]. In a hash table, chaining means that each table entry points to a linked list of values which hash to that slot. After finding the appropriate slot in the hash table, the linked list must be searched sequentially to determine if the desired data is there. This makes the size of each entry in the table variable and greatly complicates all operations on the table.

Another method for dealing with collisions is called *open addressing* [4, p. 232-6]. This method uses a more complicated hash to produce a series of values instead of just one. If the desired data is not found in the first slot, then the second value in the series is computed and that slot is checked. This continues until the data is found or an empty slot is reached. This method maintains all of the data within a fixed size table but it makes deleting values from the table extremely difficult. Since an instruction cache hash table will need to have entries removed when blocks are deallocated, this makes open addressing a poor choice.

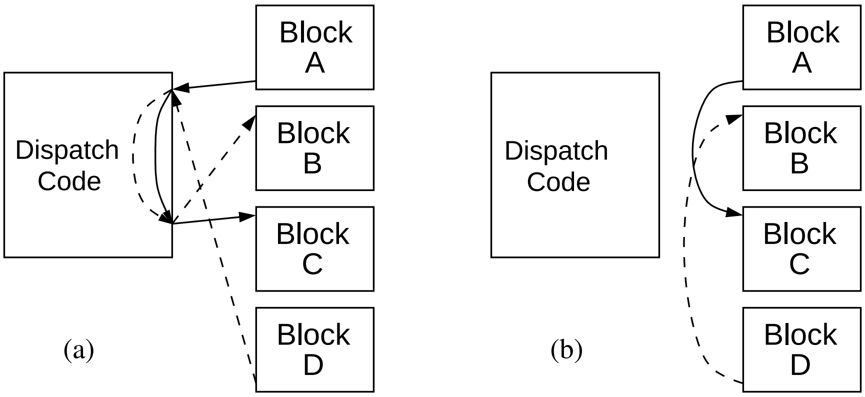


Figure 2-2: Example of jumps between blocks (a) without chaining and (b) with chaining.

Both of the previous solutions assumed that every piece of data in the table is precious, *i.e.*, once inserted, an entry must remain in the table until it is explicitly removed. An instruction cache does not require such a strict rule. If an entry is lost from the table, it means that the cache has "forgotten" that it has a certain block loaded. If that block is needed, it will simply be reloaded, incurring a performance penalty but maintaining correct functionality. This suggests a strategy for conflict resolution where the old entry is simply thrown away. As long as the hash table is large enough and the hash function is fairly uniform, this should occur infrequently and the performance degradation will be low. This is the method which was chosen for our system.

A slight modification of this scheme could provide space for two entries in each table slot. Conflicts would be handled using a chaining strategy but with a maximum chain of two elements. If more entries were hashed to that slot, the older ones would fall out of the chain. This strategy provides more flexibility for the same amount of memory as the previous solution but it also incurs a higher overhead for most cache operations which must now check both slots in the hash table entry.

2.4 Chaining

In a system with a complex caching scheme, the dispatch code could become a major bottleneck. Even if the dispatch code is fast, a small block size would cause jumps to it very frequently. Although the dispatch code is designed to be as fast and efficient as possible, it will still introduce overhead which may not be necessary. For blocks which are known to be in the cache, the program could branch directly to the desired block instead of jumping to the dispatch code. This can be accomplished using a technique called *chaining* which has been shown to produce significant benefits in the simulation systems mentioned above [11, 2]. This should not be confused with the hash table conflict strategy called “chaining” which was discussed above.

Chaining cuts out unnecessary jumps to the dispatch code by modifying the code in the cache. When the dispatch code loads a block into the cache, it goes back and replaces the jump which requested that block with a jump directly to the block. Now the next time that code is executed, it will skip the dispatch code (see Figure 2-2). This procedure will pay especially big in things like loops where the entire loop can be resident in the cache and execute many iterations without the dispatch code slowing it down. Chaining can be performed, not only when a new block is loaded, but also when a block is requested that is already present in the cache. In fact, the dispatch code can chain every time it is executed except when the original jump was an indirect jump (*i.e.*, the target address was stored in a register) [3].

The problem with chaining is that it greatly complicates deallocation. When a block that has been chained to is deallocated, the jump(s) that was modified must be changed back to a jump to the dispatch code. This is necessary to allow reloading of the block in case it is needed again later. This *dechaining* can be difficult since the normal chaining scheme does not keep track of who has chained to a certain block.

A block can be augmented with a list of the jumps that have been chained to it or a separate table of these chained jumps could be kept. However, the lists of chained jumps may be variable in size. It would be very difficult to allow these lists to change in size dynamically. The solution may be to allocate a fixed size list when the block is loaded and only allow new chainings to be performed if there is space in this list [2, p. 29]. This restricts the amount of chaining that can be done but simplifies the task of deallocation since there is now a fixed maximum number of chains to undo.

If a FIFO replacement strategy is being used, then another solution is possible. If an older block chains to a newer block then that chain will never have to be undone because the older block will always be deallocated first [2, p. 29]. In other words, it is not possible for the newer block to be deallocated until the older block has already been removed thus making dechaining a moot point. Therefore, correctness can be preserved without the bookkeeping and overhead of dechaining by only allowing chains from older blocks to newer blocks. These two solutions could also be combined by modifying the first solution to only track and undo chains from newer blocks to older blocks.

Chapter 3

System Implementation

The goal of this thesis was to design and implement a working software based instruction cache system. The initial design was to be simple and provide a framework on which future work could be based. This chapter describes the design which was chosen and discusses how it was implemented in the RAW compiler which is being developed.

3.1 Design

The system which has been designed attempts to compromise between the different design options discussed earlier. Whenever possible, block size and memory organization choices were made on the side of simplicity. Considerable effort was put into trying to design efficient data structures for maintaining the cache state but in the end, a straightforward yet potentially sub-optimal scheme was used.

3.1.1 Memory Organization

The first decision made was that the initial system would not deal with densely packing variable sized blocks. Therefore, memory is divided up into a number of fixed size slots which are managed as a heap. Because the blocks are fixed size, this is roughly equivalent to a fully associative cache with a FIFO replacement policy. A FIFO policy approximates LRU in an instruction cache but is far easier to implement since it is only necessary to keep track of the head and tail of the heap. Managing the cache as a heap makes the transition to variable sized blocks easier if that is deemed important in the future. On the other hand, using fixed size blocks means that it is not too difficult to switch to a set-associative scheme either. Therefore, this arrangement makes a good base system that can be easily modified for future research.

3.1.2 Block Size

Arbitrarily created fixed sized blocks were ruled out as a cache block size due to the increased difficulty of keeping track of entry points in the blocks. A basic block, on the other hand, has the desirable properties that there is only one entry point and no unneeded code will be loaded. It is also reasonably easy to find basic blocks within a program. It was therefore decided that basic blocks would be used for the cache blocks. Since extended basic blocks and clusters of basic blocks are really extensions of a basic block, this choice allows for a straightforward transition to one of these other block sizes in the future.

However, the variable size of basic blocks conflicts with memory organization which was chosen. Small blocks will not fill the slots while large blocks may be too big for one slot. Blocks that are smaller than a slot will waste space in the cache but will not impede the correct operation of the cache. These blocks can be padded with `NOP` instructions to make them fill a slot or they can simply be loaded into the beginning of the slot. Since all blocks will end with a jump to the dispatch code, the remaining instructions in the slot are not important.

Basic blocks that are too large to fit within a slot will overwrite the next slot if we blindly try to load them. Therefore, large blocks must be broken up into smaller ones. Since a change in control flow (*i.e.*, a branch or jump) ends a basic block, a large block can be divided up by inserting a jump into it which simply jumps to the next instruction. This will degrade performance by forcing jumps to the dispatch code even when the compiler knows that the code will be executed sequentially but it is necessary to preserve correctness.

A slot size of 16 words (equal to 16 instructions) was chosen for this system in order to balance the space wasted by small blocks with the extra overhead of breaking up large blocks. Translating a branch into a jump to the dispatch code (discussed later) adds 5 instructions to a block, meaning that a typical 16 word block can hold no more than 11 instructions from the original program. The next smallest logical slot size would have been 8 words but since this would only have allowed for 3 original instructions per block, it was deemed too small. Sixteen words is also convenient because it is the largest block transfer which can occur over the dynamic communication network in RAW.

3.1.3 Data Structures

A hash table was chosen to keep track of loaded blocks and their locations in the cache (physical addresses). This method was chosen over the array structure mentioned in Section 2.3.1 because it will work for any program, regardless of size. The array method is viewed as an optimization for small programs to be explored in the future.

Conflicts in the hash table will be resolved by discarding the data which is currently occupying the desired slot. This is not only the fastest method of resolving conflicts, it also avoids adding overhead to critical operations such as lookups. Even a limited form of chaining would require extra checks during most operations. The memory which would have been used to allow two entries in each slot will be used to add more slots to the table, thus decreasing the frequency of conflicts.

3.2 Implementation

Using the design outlined above, a compiler pass was written to implement part of a software based instruction caching system. This compiler pass is part of the `rawcc` compiler being developed using the SUIF compiler system. Because instruction caching must deal with the actual machine instructions of a program, the pass is written for the *machsuif* back-end of the compiler and is designed to be the final compiler pass. In order for the pass to have accurate information about the size of various basic blocks, all optimization passes and assembly language macro expansion passes must have already been run on the program.

The system has been implemented to the point where it runs as if all of the blocks fit and have already been loaded into the instruction memory of a single tile. The program is divided into basic blocks no bigger than 16 instructions and the code is modified to jump to the dispatch code at the end of each block. The virtual address of the block to transfer control to is passed to the dispatch code during this jump. The dispatch code looks up the virtual address in the hash table and transfers control to the physical address which is stored there. Although the dispatch code does perform the check to see if the requested block is in the cache, it does not currently handle the case when it is not. Therefore, the cache works when all of the blocks for a program have already been loaded into the instruction memory.

The portion of the system which handles cache misses has not been implemented because the simulator is not currently accurate enough. When the simulator is completed the portion of the system which deals with loading code from an external memory can be added in. In the current simulator, the entire program is placed in instruction memory (by the simulator) and then executed. Therefore, the portion of the cache system which has been implemented can be tested by arranging the program so that, when it is placed in the instruction memory, it appears as though all of the program's blocks have been loaded into slots already. For these tests, the virtual addresses of the blocks and their physical addresses in the instruction memory are the same. The missing cache functionality has been designed and will be discussed later.

3.2.1 Program Code Modifications

The compiler pass begins by modifying the existing branches and jumps to jump to the dispatch code instead of their targets. Since blocks could be loaded anywhere in the instruction memory, a jump with an absolute (rather than relative) address is needed to get to the dispatch code. If a relative address were used, it would have to be modified when the block was loaded to reflect the distance from the block to the dispatch code. On the other hand, if the dispatch code is always present in the instruction memory at a predictable location, then a jump to that absolute address will always work, regardless of where the block was placed. In the MIPS instruction set, absolute jumps are performed with the various `j` (for "jump") instructions while relative jumps are performed with the `b` (for "branch") instructions. Since the RAW instruction set is based on the MIPS instruction set [6], `j` instructions will need to be used to jump to the dispatch code.

The virtual address which control should be transferred to is passed to the dispatch code in the assembler temporary register (`$at`). This register is normally reserved for use when the assembler needs a temporary register in its expansion of a macro instruction. Since the instruction caching pass will be run after all expansions have finished, it has full knowledge of when `$at` is used and can avoid any conflicts.

The simplest jumps to replace are `jr` instructions. Instead of jumping to the value stored in the register, that value is moved to `$at` and a jump is made to the dispatch code (see Figure 3-1(a)). Almost as simple are `j` instructions. The jump is replaced with a load of the jump address into `$at` followed by a jump to dispatch (see Figure 3-1(b)). By making use of the delay slot where it previously was not, the jump to dispatch takes only one cycle more than the original jump. Jump-and-link instructions are handled similarly except that the link register must also be loaded with the address of the instruction following the modified jump.

Conditional branches are probably the most complicated jump to replace. A conditional branch can transfer control to one of two different locations. Therefore, the code which replaces one must

```
jr     $5
add    $0,$0,$0
```

↓

```
j      dispatch
add    $at,$0,$5
```

(a)

```
j      Label
add    $0,$0,$0
```

↓

```
lui    $at,Label>>16
j      dispatch
ori    $at,$at,Label&0xFFFF
```

(b)

```
  bne    $1,$2,Label1
  add    $0,$0,$0
Label2:
```

↓

```
  bne    $1,$2,New_label
  lui    $at,Label2>>16
  j      dispatch
  ori    $at,$at,Label2&0xFFFF
New_label:
  lui    $at,Label1>>16
  j      dispatch
  ori    $at,$at,Label1&0xFFFF
Label2:
```

(c)

Figure 3-1: Example replacements of (a) a `jr` instruction, (b) a `j` instruction and (c) a conditional branch instruction.

load one of two different virtual addresses into `$at` and then jump to the dispatch routine. Since only branches can be conditional and since we must use a jump to get to the dispatch code, this replacement will need to use both `b` and `j` instructions. Figure 3-1(c) shows an example of a conditional branch replacement. This code makes use of delay slots to perform the call to the dispatch code in an extra 3 cycles when the branch is taken and an extra 2 cycles when it is not.

After all of the branches and jumps have been modified, jumps to the dispatch code are inserted at the end of each block which would normally fall-through to the next block (like block A in Figure 2-1(a)). This is done with a simple 3 instruction sequence like the one in Figure 3-1(b) except that the label used is the label at the beginning of the next block.

The final step in modifying the program code is to check block sizes and break up large blocks into smaller ones[1]. All blocks are made exactly sixteen instructions long. Blocks that are larger than sixteen instructions have jumps (three instructions) inserted every thirteen instructions. When doing this, care must be taken not to insert a jump into the middle of one of the sequences created during the previous modifications. If a jump would be placed into one of these sequences, then the jump is placed right before the sequence instead. When all large blocks have been broken up, the blocks that are smaller than sixteen instructions are padded with `NOP` instructions. This is not strictly necessary in this system but was done so that the blocks would be aligned as if they had been loaded into the instruction memory by the dispatch code.

[1]Although it is conceptually clearer to do this step first, from a practical standpoint, it is easier to do it last because the other modifications add extra instructions.

3.2.2 Dispatch Code

The dispatch code is written in RAW machine language (see Appendix A) and is added to the program and the end of the compiler pass. The first action that the dispatch code must perform is a check to see if the requested block is in the cache. This involves looking up the virtual address of the block's entry point (passed to the dispatch code in `$at`) in the hash table. Although hash functions which merely select some of the bits of the address are common in cases where speed is crucial, this method does not always lead to a uniform distribution of hash values. Because conflicts cause information to be lost in this system, a function which has a better distribution of values was needed. The hash function chosen is a multiplicative function that has good performance yet is still reasonably easy to calculate. The key is multiplied by 2654435769 and then bits 23 to 31 of the result are used as the hash value. See [4, pp. 228-9] for a derivation of this function.

Nine bits are selected in order to get 512 possible hash values. This is roughly twice the number of sixteen instruction blocks which will fit in a 16 kbyte instruction memory (minus some space for the dispatch code) giving a load factor of about 0.5 for the table. Since collisions are costly, it is important to keep the load factor as low as possible. This load factor was chosen arbitrarily and may be modified in the future if it is found to be inappropriate. However, the desire for a low load factor must be balanced with the hash table's footprint. With each entry in the table needing two words of memory (one for the virtual address and one for the physical address), a 512 entry table consumes 4 kbytes of memory. This is a sizable portion of the 32 kbyte data memory and it is important to remember that increasing this table size may actually decrease overall performance by reducing the amount of memory available to cache data or instructions.

Once the virtual address has been hashed to give an index into the hash table, the dispatch code must check that hash table entry to see if the desired block is in the cache. It does this by comparing the desired virtual address with the virtual address stored in the table. If they do not match, then either that block has not been loaded yet or another block that hashed to the same value has been loaded. The table is initialized to an impossible value for the virtual addresses so that the lookup will fail the first time each entry is checked. If they match, then the desired block is available so the physical address is read from the table and a `jr` instruction is used to transfer control to it. The process repeats when execution reaches the end of that block and another call to the dispatch code is made.

If the virtual addresses fail to match then the cache miss routine is executed. In the current implementation this routine is merely a stub. Since the virtual and physical addresses are the same in the simulator and since every block is present in memory, the stub writes the virtual address of the desired block into the hash table for both the virtual and physical addresses. The stub then transfers control to the requested block via a `jr` instruction.

In the final implementation, the cache miss routine will have the job of requesting the missing block from external memory, placing the block into the instruction memory and updating the hash table to reflect its location. Because code execution is dynamic in nature, the dynamic communication network will need to be used for requesting data from off-chip. The tiles around the perimeter of the chip will have an I/O interface to external DRAM so requesting data from off-chip involves sending a message to one of these perimeter tiles. The request message will simply be composed of the starting address of the block in external memory, the amount of data needed (sixteen words) and the tile number to return the data to. The miss routine must now wait for the data to be sent back to it. During this time, it can select the location for the new data and update the hash table and pointers for the beginning and end of the FIFO queue of blocks. When the data actually arrives, it is simply copied into the selected space in the instruction memory and control is transferred to it.

3.3 Results

The instruction caching compiler pass was run on several benchmark programs to evaluate performance. Because the implementation handles only the cache hit cases, benchmarks which fit entirely within the instruction memory were chosen[2]. Therefore, the performance numbers collected indicate the minimum amount of overhead needed during the execution of these programs with software instruction caching. Additional overhead will be incurred due to cache misses in the final system.

Because the benchmarks used all fit within the instruction memory, they do not actually need to use caching at all. Normally the compiler would detect this and omit instruction caching from the program. The results of running the programs without caching added are given as “uncached” in Table 3.1 and Table 3.2. In order to evaluate the impact of caching on programs, the compiler was then forced to add the caching code to these benchmarks. The results of running with the currently implemented system are reported as “fixed size.”

As a beneficial side-effect of the incomplete implementation, one more case could be tested.

[2]In reality, the current simulator does not limit the amount of instruction memory available. Therefore, any program will fit entirely within instruction memory.

	uncached		variable size		fixed size	
benchmark	cycles	change	cycles	change	cycles	change
life	1,302,132	1.0x	3,665,808	2.8x	4,938,266	3.8x
jacobi	1,485,328	1.0x	2,589,359	1.7x	4,636,892	3.1x
vpenta	15,173,695	1.0x	18,035,420	1.2x	36,275,951	2.4x
cholesky	24,879,626	1.0x	47,243,774	1.9x	77,263,030	3.1x
tomcatv	63,847,647	1.0x	94,787,869	1.5x	164,020,150	2.6x
btrix	111,014,163	1.0x	134,755,597	1.2x	269,574,949	2.4x

Table 3.1: Run time for various benchmarks (in processor cycles) without any caching, using variable sized blocks and using fixed size blocks. The "change" column is relative to the uncached version.

	uncached		variable size		fixed size	
benchmark	memory	change	memory	change	memory	change
jacobi	604	1.0x	960	1.6x	1664	2.8x
life	1248	1.0x	2064	1.7x	8392	6.7x
cholesky	2544	1.0x	4292	1.7x	7572	3.0x
vpenta	3780	1.0x	4436	1.2x	6364	1.7x
tomcatv	4624	1.0x	6220	1.3x	9756	2.1x
btrix	15128	1.0x	16504	1.1x	21884	1.4x

Table 3.2: Bytes of instruction memory used by the program, excluding the dispatch code and hash table. The "change" column is relative to the uncached version.

Because all of the code for these programs is already in memory and because blocks are never deallocated from the cache, it is possible to simulate using densely packed variable sized blocks. By not breaking large blocks up into sixteen instruction blocks, the execution time of a cache using variable sized blocks can be measured. The amount of memory that would be consumed by a variable sized block system can be determined by neither breaking up nor padding blocks. The numbers from simulated variable sized block caches are given as "variable size" in the tables.

Looking at the data, adding instruction caching to a program clearly has a significant impact on both performance and memory usage. However, the penalty is substantially less for the variable sized blocks than it is for the current implementation's fixed size blocks. It should also be noted that the penalties for both schemes tend to be less for the programs which are larger or run longer.

The most surprising result found was the tremendous difference between the variable sized block and fixed size block schemes. Both performance and memory usage were significantly worse when using a fixed size block. The difference in memory usage is primarily due to small blocks which waste space when loaded in sixteen instruction slots. However, there is also some memory lost to the extra jumps which are inserted to break up large blocks. The performance difference is explained by the extra calls to the dispatch code which are created when large blocks are broken into smaller ones. In the benchmarks used, the largest blocks tend to occur in the middle of nested loops. Unfortunately, this means that the extra overhead is greatly amplified by the fact that those blocks are executed many times.

Although the impact on program performance can be quite large, it is important to remember that this system is designed to be used only on programs that do not fit within the instruction memory. For these programs, the performance using caching would be infinitely better than without since they cannot be run at all without instruction caching. Even so, it is important to minimize the overhead of caching in order to compete with hardware based caches.

Chapter 4

Conclusions

Although this system is a starting point rather than a final solution for all software based instruction caching needs, it is possible to draw some conclusions about software based instruction caching and make suggestions for future work.

4.1 Future Work

Clearly the next step for this system would be the addition of the cache miss handler. With that in place, hash table performance should be carefully examined. If collisions occur frequently, it may be desirable to modify the hash table size or the collision handling strategy. It might even be necessary to develop a new hash function which is more specific to the access patterns of this system. In addition, using other data structures which incur less overhead for smaller programs (as described in Section 2.3.1) is key to taking full advantage of the compiler and will definitely need to be pursued.

Based on the results given in Section 3.3 it is apparent that a variable sized block scheme is highly advantageous. Not only does it immediately increase performance, but the reduced memory footprint will also allow more blocks to fit into memory, thus increasing the cache hit rate. Since the benefit for the common case (where the desired block is present in the cache) is so great, it seems likely that it would be worth the extra overhead associated with a variable sized block system. In fact, the extra overhead of such a system would occur in the cache miss handler which is likely to spend large amounts of time waiting for new data to arrive. It may turn out that the extra overhead can be overlapped with this waiting, thus making it free. Such a system should definitely be implemented to explore its feasibility.

Since the amount of time it takes data to be fetched from external DRAM and returned to a tile may be large, complex schemes for placing and deallocating blocks could be used without impacting overall performance. Depending on how much time a fetch takes, it may actually be possible to implement a replacement strategy which approximates LRU. Of course, reconciling this with a variable block size would still be difficult. Even so, the possibility should be explored.

Even using a variable block size, performance was significantly affected. Since this result is the minimum overhead of the complete system, this suggests that chaining may be a worthwhile optimization. Again, the extra overhead would be added while waiting for data from external memory so the impact could be negligible. The mostly likely implementation of chaining would include one or two back pointers for each block and would only keep track of backwards chains as discussed in Section 2.4.

The last variable which probably warrants exploration is block size. Given that the frequent jumps to the dispatch code do significantly decrease performance, it is probably wise to use a larger block size. Of the options discussed in Section 2.1, clusters of basic blocks show the most promise for increasing performance. However, because clusters of basic blocks can load code which is never used, the cache hit rate will be lower. It will be necessary to determine whether the extra performance offsets the reduction in cache hit rate. In addition, there is work to be done in developing a compiler to find optimal clusters.

4.2 Conclusion

This work suggests that software based instruction caching may be a viable alternative to hardware instruction caching. Even in this naive implementation, the added overhead for the common case is not prohibitive. Exploration of the many optimizations and alterations which are possible will undoubtedly yield systems with even better performance.

Software based caching provides RAW with the ability adapt to different workloads. The resources devoted to caching can be varied to fit the requirements of a specific application. However, software based caching is also applicable to low-cost or low-power embedded processors where the cost of caching hardware is prohibitive. With further research and time, software based caches may become common-place, even replacing hardware caches in general purpose microprocessors.

Appendix A

Dispatch Code

```
dispatch:
        sw      $9,save_t1
        lui     $9,40503           # Load the hash constant into $9
        ori     $9,$9,31161
        multu   $9,$at             # Multiply the key by the hash constant
        sw      $10,save_t2
        mflo    $9                 # Select bits <31:23> of the result
        srl     $9,$9,23
        sll     $9,$9,3            # Scale for the size of each table entry
        lw      $10,hash_table($9)
        bne     $at,$10,dispatch.miss   # Compare virtual addresses
        add     $0,$0,$0
        addi    $9,$9,4                 # Cache hit!
        lw      $at,hash_table($9)      # Read physical address from table
        lw      $9,save_t1
        jr      $at                     # Jump to the requested block
        lw      $10,save_t2
dispatch.miss:
        # This is a stub which fixes up the hash table.  It will
        # be replaced by code to load a new block into the cache.
        sw      $at,hash_table($9)      # Enter virtual address in table
        addi    $9,$9,4
        sw      $at,hash_table($9)      # Physical address is the same
        lw      $9,save_t1
        jr      $at
        lw      $10,save_t2
        .end    dispatch
```

References

[1] A. Agarwal, S. Amarasinghe, R. Barua, M. Frank, W. Lee, V. Sarkar, S. Devabhaktuni, and M. Taylor, "The Raw Compiler Project", *Proceedings of the Second SUIF Compiler Workshop*, Aug. 1997.

[2] R. F. Cmelik and D. Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling," SMLI 93-12, UWCSE 93-06-06, Sun Microsystems Laboratories, Inc., and the University of Washington, 1993.

[3] R. F. Cmelik and D. Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling," *Proceedings of the Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pp. 128-137, May 1994.

[4] T. Cormen, C. Leiserson and R. Rivest, *Introduction to Algorithms*, The MIT Press, Cambridge, Massachusetts, 1996.

[5] D. R. Engler, "VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System", *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pp. 160-170, May 1996.

[6] J. Heinrich, *MIPS R4000 Microprocessor User's Manual*, MIPS Technologies, Mountain View, California, 1994.

[7] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, San Francisco, California, 1996.

[8] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar and S. Amarasinghe, "Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine", *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, Oct. 1998.

[9] E. Waingold, M. Taylor, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktuni, R. Barua, J.Babb, S. Amarasinghe and Anant Agarwal, "Baring It All To Software: Raw Machines", *IEEE Computer*, pp. 86-93, Sept. 1997.

[10] S. Ward and R. Halstead, *Computation Structures*, The MIT Press, Cambridge, Massachusetts, 1990.

[11] E. Witchel and M. Rosenblum, "Embra: Fast and Flexible Machine Simulation," *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 68-79, May 1996.