Compiler Controlled Cache Placement

David Koes
dkoes@cs.cmu.edu

David McWherter
cache@cs.cmu.edu

December 15, 2002

Abstract

This paper considers a novel approach for supporting parallel accesses to a data cache. We explore the possibility of explicitly managing cache accesses using a static compile-time analysis. First we perform a limit study on dynamic instruction traces to discover an upper bound on the amount of memory parallelism that is exploitable at compile time. Then we introduce *compiler controlled split caches*, or cc-split caches, and discuss our initial implementation. Our results suggest this architecture has potential, but we question whether moving complexity from the hardware to software is a profitable trade-off in this case.

1 Introduction

Modern processors are being designed to exploit an increasing amount of parallelism. As a result, modern processors require higher bandwidth access to data caches. Three currently implemented types of caches which provide parallel access are:

- *Ideal multi-port.* A true multi-ported cache, such as the Intel Itanium's dual ported L1 data cache[9] provides the best possible performance. However, this scheme does not scale well in the number of ports and requires a prohibitive amount chip area relative to the cache size. For example, an ideal dual ported cache requires more than twice the chip area as a single ported cache.

- *Copy Cache.* A copy cache maintains a copy of the cache for each memory port in the system. When a load is issued, it is dispatched to the first available memory port. Every store must be issued simultaneously to each cache copy in order to maintain coherence. Since a copy of the cache is maintained for each memory port, the chip area required to implement this scheme scales very poorly with respect to the number of memory ports. However, the logic required to implement a copy cache is relatively simple. The DEC Alpha 21164 has an 8k dual ported L1 data cache which is implemented with two 8k copies of the cache[8].

- *Multi-banked cache.* In a multi-banked cache, the cache is split into multiple banks each of which can be accessed in parallel. A load or store is dispatched to a certain bank based on the value of the address of the load or store. The cost of implementing the crossbar which routes requests between the processor and the cache is minor relative to the size of the cache. Increasing the number of banks does result in a larger crossbar which may increase the cache access time. However, it has been shown that the biggest performance bottleneck for multi-banked caches are the bank conflicts that occur when two memory operands try to access the same bank in parallel[13]. As the number of banks is increased, the performance of the system improves with only a small increase in required chip area[10]. The Intel Pentium utilizes an 8 bank data cache[7].

The purpose of this study is to is to explore the ability to provide enhanced memory parallelism without using complex and expensive hardware by allowing the compiler to make static choices regarding cache bank selection. First, we analyze the memory access behavior of various programs by examining dynamic instruction traces. We describe our approach to this limit study in section 2 and our results in section 3. In section 4 we describe our *compiler controlled split cache* scheme which allows the compiler to explicitly managed cache bank selection and then give our results in section 5. Finally we describe future work and conclude in section 6.

2 Dynamic Analysis

Part of the overall goal of our work is to understand whether static placement of data into the banks of a banked cache would improve overall performance. Through the examination of dynamic traces an up-

per bound on the improvement that can be achieved using static placement can be determined.

We make the assumption that the mechanism for static bank placement involves tagging individual load and store instructions at compile time to indicate which cache banks should be used. Under this assumption, any static placement algorithm must ensure that every instruction that accesses a given memory address be tagged in the same way. Otherwise, the results of the program may be inconsistent with those using a single-banked cache. Consider, for example, a sequence of loads and stores as follows:

Op	**Address**	**Value**	**Cache Bank**
load	0x4	0	A
store	0x4	1	B
load	0x4	0	A

Since cache bank A was not updated during the store through cache bank B, the final load results in an incorrect value.

Given the assumption that load/store instructions specify which cache banks are used, we can examine dynamic traces of programs' memory accesses to determine the optimal assignment of banks to instructions for those traces. Whenever two instructions happen to access the same address, they must be assigned to the same bank. Otherwise, they are may be placed into different cache banks, which can improve performance.

Clearly, the assignment of banks to instructions done according to the dynamic trace will be better that the assignment that a compiler would ever be able to generate. Due to aliasing and other problems, a compiler will often not be able to determine that two instructions never access the same data. As a result, a compiler will be forced to conservatively label two instructions with the same cache bank even though they never access the same data at runtime. Analysis using dynamic traces is much easier than implementing sophisticated alias analysis in a compiler and provides a good picture of the limit of what static analysis could achieve.

2.1 Methodology

In order to collect dynamic memory access traces from benchmark applications, we made use of the Simplescalar simulator[3]. Simplescalar simulates a very reasonable RISC processor derived from the MIPS-IV ISA, suiting it for modern architectural studies. In addition, there is a large amount of support for the architecture, including ports of binutils, gcc, and many spec benchmarks for the architecture, making it convenient to use.

We modified the Simplescalar simulator to keep track of every memory access made as the system executes. We track every memory address accessed along with the program counter of the instruction that accessed the memory. For each memory access we form a $<address, pc>$ tuple The least significant bits of the address which are used to index into the cache line are ignored. For our analysis, we use a cache line size of 64 bytes. These tuples are used to construct a mapping of addresses to lists of program counters which access that address in the dynamic trace, or $<address, pclist>$ tuples.

Due to the sheer quantity of $<address, pc>$ tuples generated during the execution of a normal program, and the desire to hack Simplescalar as little as possible, Simplescalar was modified to write these tuples to a secondary application through a named pipe. This secondary program collected and collated the data, eliminating duplicate tuples, to reduce the storage requirements of the data.

Given a set of $<address, pclist>$ tuples generated during the execution of an application, we are left with the task of determining which of the instructions must be forced to use the same cache banks, due to accessing the same data. This can be computed fairly efficiently using a disjoint set algorithm (disjoint set with path compression). First, we make a set for each program counter seen in the data. Then, for each $<address, pclist>$ tuple, we union all of the program counters in the *pclist* together.

For example, assume program counter $PC1$ accesses addresses $\{A, B\}$ and $PC2$ accesses addresses $\{B, C\}$, and $PC3$ accesses address C. After the first stage of post processing, we will have the following tuples:

Address	**PC List**
A	PC1
B	PC1, PC2
C	PC2, PC3

During the disjoint set algorithm, we will call both $union(PC1, PC2)$ and $union(PC2, PC3)$, placing all of the program counters into the same set. Note that this is despite the fact that $PC3$ doesn't access any of the same data that $PC1$ accesses.

This property, that instructions that don't directly access the same data $(PC1, PC3)$ may be forced into the same bank due to interactions through other program counters $(PC2)$ seems to be the largest problem for static bank placement. The reason is that as the number of sets in the disjoint set decreases, the ability to find a good assignment of banks becomes more difficult. At one extreme, when there is only one large set, all of the banks save one will be left unused in the

system. At the other extreme, when every program counter is in a different set, we can split data evenly between sets without any difficulty.

2.2 Stack Accesses

Another problem associated with this approach is that stack data may cause a lot of "false sharing" between various program counters. When a function is called at some location on the stack, its stack manipulation instructions will join with all other functions' instructions that are called at that location as well. Local stack data that is not passed to child functions does not need to be kept consistent with reads and writes across other functions as long as each function's stack area doesn't share cache lines with other stack areas. As a result, it would be reasonable to assume that these accesses should not be used to union PC sets together.

Stack data passed to child functions, however, needs to be accessed in the right cache bank to ensure correctness. In the simulator, it is more difficult to detect whether a memory access to the stack is to the local stack, or to the stack frame of another function. This, combined with our desire to upper bound our ability to assign banks to instructions, leads us to assume that accesses to stack memory never unions program counter sets. Thus, even if the address of stack data is passed as an argument to a function, accesses to that data in the child will not join sets.

2.3 Inlining

If one thinks carefully about what causes the joining of sets of instructions in the disjoint set, it is not hard to see that the lack of inlining of function calls could increase the number of unions that must be done. Assume that function $fooA()$ accesses array A, and $fooB()$ accesses array B and the arrays do not alias. Let $Inst(A)$ be the instructions outside of $memset()$ that access array A and likewise for $Inst(B)$. If $fooA()$ calls $memset(A)$ and $fooB()$ calls $memset(B)$, then the disjoint set will result in a large set containing $\{memset(), Inst(A), Inst(B)\}$.

If, on the other hand, $memset()$ had been inlined into $fooA()$ and $fooB()$, then we would have been left with two independent sets, $\{memset.fooA(), Inst(A)\}$ and $\{memset.fooB(), Inst(B)\}$. Thus, if we assume that the compiler aggressively inlines functions, the number of sets should increase dramatically — and likewise, the possibility of finding an ideal assignment of banks to instructions.

Our compiler, however, is inconsistent with inlining of functions, even with library functions such as $memset()$ or $memcpy()$. In order to determine how much an aggressively inlining compiler could help set construction, we decided to simulate the behavior of inlining when constructing the disjoint set. In order to do this, we modified the Simplescalar simulator to keep a stack of the program counters of branch-and-link instructions used to make function calls. Thus, for each memory access, we keep track of $< address, parentpc, pc >$ tuples. By building a disjoint set of $< parentpc, pc >$ pairs that access the same memory addresses, we effectively perform one level of inlining everywhere in the code. Thus, if $memcpy()$ failed to be inlined by the compiler, and it is called by $fooA()$ and $fooB()$, we will have entries for $fooA.memcpy()$ and $fooB.memcpy()$ in the disjoint set.

This approach is very similar to automatically inlining every function. The technique will, however, often produce results that are "more aggressive" than inlining. The reason is that we ignore such things as cycles in the call graph, which obviously cannot be fully inlined. We, however record each member of the cycle as being inlined in its parent.

We would expect this technique to increase the number of sets in the disjoint set, and perhaps make assignment of banks to instructions a much easier task.

2.4 Inter-PC-Set Contention

Regardless of whether we can generate a large number of PC sets or not, it becomes important to assign a cache bank to each set. The goal in assigning cache banks is to reduce contention for memory ports on the caches. Formally, we define cache bank contention as the event that a load or store must wait in the Load/Store Queue because it cannot be serviced due to the lack of a free memory port.

In order to reduce bank contention, our goal is to assign banks to PC sets so as to minimize the number of remaining contentions. Note that assigning different banks to two contending instructions will remove that contention from the system, as they will be serviced at different cache banks, each with their own memory port. Since we cannot break contentions between instructions within the same set without allocated to multiple banks, we ignore intra-set contentions. Furthermore, we should note that assignment of banks to sets may enable the elimination of some contention, but may prevent the removal of other contentions as the banks we choose are fixed. The problem is one of finding the maximal k-coloring of the contention graph.

Due to the difficulty of the coloring problem and the discouraging results of our contention data, we

did not examine heuristics or algorithms for assigning banks to these PC sets.

3 Limit Study Results

3.1 PC Sets

Num Sets	Size	% of MemInsns
1	2742 PCs	95.4%
34	6–11 PCs	1.23%
97	1–5 PCs	3.37%

Table 1: Distribution of memory instructions (PCs) between disjoint memory sets in `compress` with stack addresses. Almost everything is stuck into one big set.

The construction of PC sets by joining instructions based on stack references is particularly troublesome. For all of our benchmark applications, `ijpeg`, `compress`, and `li`, almost all of the instructions are put into one large set. Table 1 shows this behavior for the `compress` benchmark. We found 57 sets, and 95% of the instructions wound up being placed into one of those sets.

This is particularly worrying due to the simple fact that any static placement algorithm will have very little leeway in choosing banks for a program's instructions. The 95% of instructions must all be placed in one cache bank. In a banked cache system, this would result in horrible cache utilization, making the mechanism undesirable.

3.2 Stack Removal

We can make the assumption that accesses to the stack are able to be handled specially, and should not be used to join sets of program counters. Analysis reveals that this assumption increases the number of sets found in the disjoint set. In `compress`, for instance, we increase from finding 57 sets when including stack accesses, to 308 sets when excluding them from the disjoint set analysis.

Num Sets	Size	% of MemInsns
1	2237 PCs	77%
16	6–83 PCs	9.82%
35	2–6 PCs	4.9%
238	1 PCs	8.28%

Table 2: Distribution of memory instructions (PCs) between disjoint memory sets in `compress` ignoring stack addresses. Almost everything is still stuck into one big set.

Num Sets	Size	% of MemInsns
1	4778 PCs	86.85%
119	4–17 PCs	2.16%
21	3 PCs	1.145%
32	2 PCs	1.16%
492	1 PC	8.9%

Table 3: Distribution of memory instructions (PCs) between disjoint memory sets in `li`, ignoring stack accesses. Almost everything is stuck into one big set.

App	Static	Dynamic
`compress`	77.8%	99.7%
`ijpeg`	98.2%	98.67%
`li`	%	%

Table 4: Sizes of the dominating set in terms of static and dynamic memory instruction footprint. The dominating set is all that matters, statically or dynamically.

Unfortunately, the resulting sets still have the same problem as the sets computed using stack addresses. Table 2 illustrates the results for `compress`. A single large component dominates and contains almost all of the instructions. There is some hope, however, in that this large set has become much smaller, reducing from containing 95% of all memory instructions to only 77% of memory instructions.

Table 3 shows the sizes of the sets found in the application `li` when ignoring stack addresses. There were 682 sets discovered in the analysis, but 1 contained 86.85% of the instructions. The trend of a dominating set appears throughout all of our benchmarks.

The numbers above suggest that there is consistently a large disjoint set that contains most instructions. Another interesting question is how much these dominating sets are represented during the dynamic execution of the program. We counted the number of times a memory instruction from each set was executed while running the program. Table 4 shows the results of this experiment. For all of our applications, the dominating set not only is the most significant in the static set of instructions, but it is also by far the only set that matters during dynamic execution.

3.3 Inlining

Given the assumption that inlining function calls should enable us to find many more disjoint PC sets in the resulting program, we analyzed the effects that inlining had under our benchmark applications.

Under the `li` benchmark, we managed to find 682 disjoint sets of program counters under our standard pc set analysis when ignoring stack data. When also using our inlining technique, we managed to find 5001 disjoint sets. This improvement of over 7-fold is quite impressive, and would suggest that we have a much greater ability to efficiently assign banks to memory instructions.

Unfortunately, we still have the same fundamental problem that we encountered without inlining. By aggressively inlining functions, we've bloated the code. The number of memory instructions increased from 5141 instructions to 21996 instructions. Correspondingly, we observed an increase in the absolute number of disjoint sets. At the same time, the size of the dominating large set expanded to cover 16851 instructions, or 76.6% of the memory instructions. Thus, we are still stuck with the problem of having to place a large set of data into one bank of our cache, which will reduce our cache utilization.

We also examined the effect that inlining had on the `ijpeg` benchmark. Without inlining, we found 37 disjoint sets, and one large set that hit 5821 of 5927, or 98% of the memory instructions. After inlining, we found 2449 disjoint sets, by bloating the code by only a factor of two. The largest set in the inlined disjoint set contained 67.68% of the memory instructions, which is still a sizeable number of instructions.

On one hand, inlining seems as if it should be an effective solution for increasing the number of sets for bank assignment in order to make assignment easier. On the other hand, it appears that aggressive inlining fails to greatly improve the situation, as the PC sets are still dominated by a huge component, which cannot be broken and distributed across the cache banks.

3.4 PC Set Contention

The event that there is contention between an instruction in one PC set and another instruction in the same set is particularly problematic. The reason is that the set of instructions cannot be split up and be made to use different cache banks.

We examined the degree of inter- versus intra- PC-set contention on our suite of benchmarks. We found that most of the "static" contention that occurs in the benchmark programs occurs between members of the same set. Table 5 depicts the amount of intra-set contention that was found between sets in our benchmark suite under varying issue widths.

Issue Width	App	IntraSet
4	`compress`	100.0%
	`ijpeg`	54.19%
	`li`	78%
8	`compress`	100.0%
	`ijpeg`	57.9%
	`li`	69.4%
16	`compress`	100.0%
	`ijpeg`	100.0%
	`li`	75.2%

Table 5: Intra- and Inter- PC set contention. Most contention appears between instructions within the same set — in particular, between the dominating set and itself.

3.5 Inter-Function, Intra-Function Contention

When considering the implementation of a compiler that performs static assignment of instructions to cache banks, there are a number of issues that become important. One, in particular, is the computational difficulty of determining where contentions occur in the program source code.

It is a well known fact that inter-function analysis is an extremely difficult problem for modern compilers [12]. This would suggest that locating and dealing with contention across functional boundaries would be extremely difficult at compile time as well. Within a single function, contention can be approximated using heuristics if the compiler knows the cycle counts for accessing the caches. If most contention is found to be between instructions in the same function, then just "simple" intra-function contention analysis should improve performance greatly.

We measured contention between instructions in the `ijpeg`, `compress95`, and `li` spec benchmark programs under Simplescalar using 4-, 8-, and 16-issue widths (precisely, we set the parameters $issue : width$, $decode : width$, $fetch : ifqsize$, $lsq : size$ and $ruu : size$ to the desired "issue width").

First, we examined contention between instructions "statically." In order to measure this, we determined whenever two instructions contended for a memory port at runtime, and counted this contention once. This corresponds to the compiler recognizing that contention is possible, but not recognizing how many times those two instructions will contend during the execution of the program. Table 6 lists the results of this static contention analysis. Figure 1 illustrates the ratio of inter-function to intra-function static contention broken down according to application, and Figure 2 shows the ratio for `li` as the issue

Issue Width	App	IntraFn contention
4	ijpeg	97.05%
	compress	95.59%
	li	89.75%
8	ijpeg	93.946%
	compress	91.94%
	li	73.39%
16	ijpeg	87.87%
	compress	83.6%
	li	39%

Table 6: Static Intra- vs. Inter-function contention on spec benchmarks as the issue width increases. As issue width increases, more and more contention crosses function boundaries.

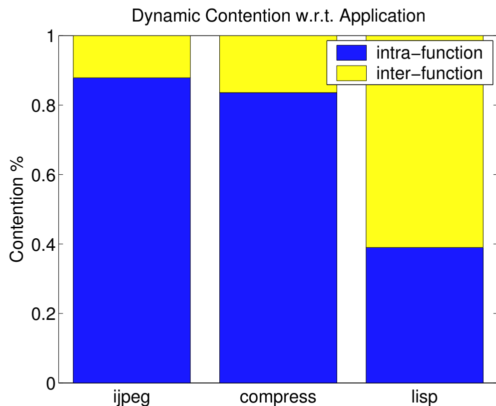


Figure 1: Static Intra- vs Inter-function contention for `ijpeg`, `compress`, and `li`.

width increases.

The data clearly indicates that for low issue widths, the number of intra-function contentions dominate the contentions present for all applications. On the other hand, as issue widths increase, the number of inter-functional contentions grows significantly, especially for the `li` benchmark. The observed inter-function contentions occur mostly between stack and frame setup, global pointer initialization, and register save/restore code. However, there are instances of contention between regular, user originating memory operations and there are even instances of contention spanning multiple functions on a 16-issue processor.

In addition to considering inter- and intra-function contention statically, we considered how things would change if dynamic behavior were taken into consideration. Thus, each time that an instruction contends with another, it is counted, and reflects the number of times the contention appears at runtime. This information is much less likely to be readily available to

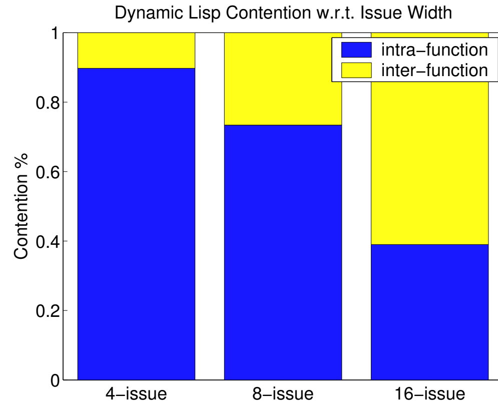


Figure 2: Static Intra- vs Inter-function contention for `li`, as issue increases. As issue width increases, inter-function contention becomes more significant.

Issue Width	App	IntraFn contention
16	ijpeg	96%
	compress	81%

Table 7: Dynamic Intra- vs. Inter-function contention on spec benchmarks. Dynamically, more than 80% of contention occurs intra-procedurally. Even `li`, which had little static intra-function contention has much dynamic intra-function contention.

a compiler without much nontrivial analysis, but it seems promising. Table 7 illustrates the dynamic contention results for the benchmark applications under 16-issue width. We find that considering runtime behavior, by far most contention occurs intra-function. Even the `li` benchmark, which had much less intra-function contention in the static analysis, has much more dynamic contention. As a result, it appears likely that intra-function contention analysis alone may be a very powerful tool in the construction of static bank assignments. Additionally, writing compiler technology to get large wins may be quite feasible.

4 Split Caches

In a *compiler controlled split cache*, or cc-split cache, the compiler can statically assign memory operations to specific cache banks. The compiler may assign a memory operation to multiple cache banks. Every load and store instruction includes a bitmask which specifies what cache bank(s) that memory operation's request will be routed to. If the bitmask is zero, then the request is dynamically routed just as it would be in a multi-banked cache.

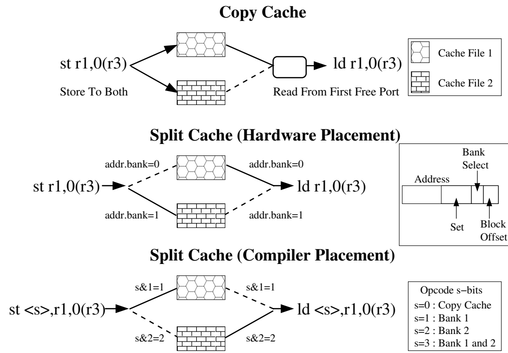


Figure 3: Different methods of cache bank selection and utilization.

When servicing a load, the processor will use the first available cache bank whose corresponding bit is set in the bitmask. When servicing a store, the processor will simultaneously store to every cache bank whose corresponding bit is set in the bitmask. If any required cache banks are not available (because they are busy servicing other requests), the memory operation will block. Note that it is possible for a cc-split cache to exhibit the full semantics of either a copy cache or a multi-bank cache. Our dynamic routing function uses bit selection.

The least significant bits of the address which aren't part of the cache line offset are used to select which cache bank the address gets routed to. The cache bank selector bits must be distinct from the bits in the address used to index into the cache bank to avoid under-utilization of cache resources. Other bank selection functions, such as XOR-folding, have been proposed but only result in marginal improvement and increase the required chip area for the routing crossbar[11][1][14].

An important side-effect of supporting dynamic routing in a cc-split cache is that the same address will map to a different cache line depending on whether it is dynamically or statically routed, even if it is routed to the same cache bank.

The compiler is responsible for ensuring the correctness of the cache bank assignments. This is absolutely necessary for good performance since the overhead involved in having the hardware maintain cache bank consistency would more than offset the advantages of a cc-split cache.

Even with the correctness constraint, there is still ample opportunity for the compiler to improve memory parallelism. When the compiler detects contention for access to memory (for example, two consecutive loads) and can determine that the memory references will never alias, it can assign the memory accesses to separate cache banks. If the compiler detects contention between memory accesses which do alias, it can still improve parallelism by assigning the memory accesses to two cache banks. This can be beneficial if increased load parallelism offsets the penalty of having to store to multiple cache banks.

We estimate that a cc-split cache would require about as much chip area as a traditional multi-bank cache. As with a multi-banked cache, the cache banks have no need to communicate among themselves and so can be physically separate.

4.1 Implementation

We have performed an initial evaluation of the cc-split cache architecture by extending the SimpleScalar 3.0 toolset. Every load and store instruction is annotated with a 16 bit bitmask as in the following example:

```
lw/15:0(2) $2,c
sw/15:0(10) $2,20($fp)
lw/15:0(4) $2,b
lw/15:0(8) $3,x.2
```

In this example there are four cache banks. The decimal number within the parentheses is the value the 16 bit bitmask is set to. Each load is assigned to a distinct, non-contending, bank. The store to the stack is assigned to two banks since this memory location is found to contend with itself later in the program.

We have modified the SimpleScalar compiler, which is based on GCC 2.7.2, to generate cache bank selection annotations. Our annotation pass happens late in the compilation after all other optimization passes. The compiler first uses the GCC 2.7.2 alias detection to build up disjoint sets of aliasing memory operands. Memory operations that are in the same set may potentially alias and therefore must be assigned to the same cache bank. Next, the compiler scans through the instruction stream and attempts to determine areas of memory contention. If two memory operands are accessed within an appropriately sized fixed window they are considered to contend unless these accesses are dissected by a possible change in control flow (such as a call instruction). Since in our implementation the cache has separate read and write ports, stores are not considered to contend with loads and vice versa.

The result of finding the contentions between memory operands is a contention graph between memory sets. An edge exists between memory sets if it is possible that two memory operations might be trying to access these two sets during the same cycle. Some memory sets (in fact, many of them) will contend with themselves. The compiler attempts to "color" all the memory sets which do not contend with a single cache bank "color" using a heuristic driven graph coloring algorithm. Then the compiler attempts to

assign self-conflicting memory sets to multiple cache banks, but only if it can do so without creating contention against other, already assigned, memory sets. When choosing cache banks, the compiler attempts to balance the number of accesses assigned to each bank when possible. If the compiler cannot successfully color a memory set it defaults to either full multi-bank mode (algorithm 1) or full copy cache mode (algorithm 2).

4.2 Limitations

There are several limitations in our current implementation affecting both correctness and performance. Since the compiler works on each function individually, it is not currently possible to ensure correctness across function calls. For example, if a reference to a global variable is assigned to different cache banks in different functions, cache consistency issues might arise if the corresponding cache line doesn't get evicted between accesses in the different functions. Some possible solutions are supporting interprocedural analysis in the compiler and linker and/or adding additional instructions for explicit cache management (allowing the compiler to force an eviction if necessary). Another serious correctness limitation not addressed by our implementation is that the compiler must not only correctly determine which memory accesses do not alias, but also make sure these accesses do not share a cache line. That is, the compiler needs to have full knowledge of the layout of data in memory.

The SimpleScalar toolset simulates cache latencies, but does not actually simulate the cache data flow (it just keeps tracks of the cache tags). As such, the expected correctness issues do not inhibit implementing an incorrect, but substantially less complicated, initial compiler infrastructure for evaluation purposes.

Our current compiler infrastructure merely annotates the conventional output of the compiler. We could further enhance performance by performing code transformations that would encourage better cache bank placement such as modulo unrolling[2], cache driven data layout[4] and structure layout[6][5], and cache bank aware instruction scheduling. Another major limitation of the current compiler is that it uses the GCC 2.7.2 alias detection routines which are quite primitive compared to modern alias analyses[12].

5 Results

To evaluate the cc-split cache architecture we compiled and ran programs from the SpecInt95 benchmark suite. All programs were compiled with our modified GCC 2.7.2 compiler using the -O2 option. The simulator is configured to simulate a full featured 4-way out-of-order superscalar processor. The number of memory ports available to the processor is set equal to the number of cache banks being simulated. The L2 cache is a unified 1MB 4-way set associative cache with 64 byte cache lines. The L1 instruction cache is a 16k direct mapped cache with 32 byte cache lines. The L1 data cache is a 32k direct mapped cc-split cache (the sum of the sizes of the cache banks is 32k) with 32 byte cache lines.

In Figure 4 we show our results for a system configured to have two memory ports and two cache banks. We compare an ideal dual port 16k cache, a pure copy cache, a pure multi-banked cache, and our two algorithms. All results are normalized against an ideal dual port 32k cache and longer bars are slower. We compare algorithm 1 against the multi-banked cache since the difference between them is caused by the compiler's ability to statically select cache banks (when the compiler is unsuccessful in selecting cache banks for a memory operation, it defaults to multi-bank cache mode in algorithm 1). For the same reason, we compare algorithm 2 to the copy cache.

In most cases algorithm 1's performance closely follows that of the multi-bank cache. In `vortex`, however, the compiler static placement is 3.17% faster. In this benchmark, the compiler managed to successfully introduce memory parallelism in a few small, very frequently called functions. Algorithm 2 does at least slightly better than the copy cache in all cases but `gcc` and `li`. The best improvement seen for algorithm 2 over copy cache is in `compress`, where algorithm 2 is 1.04% faster, but algorithm 2 is 1.69% slower on `li`. However, in most case, the copy cache and algorithm 2 don't match the performance of the multi-bank cache or algorithm 1. In the `go` benchmark the multi-banked cache and algorithm 1 actual beat the ideal 16k cache indicated that cache space is more important than perfect memory parallelism in this benchmark.

In Figure 5 we explore the effect of increasing the number of cache banks on the linpack benchmark. As the number of cache banks increases, the multi-bank cache does better since the number of bank conflicts drops. However, as the number of cache banks increases, the copy cache does worse since the amount of actual cache space decreases and the effect of having to issue multiple stores is more pronounced. Algorithm 1 manages to be competitive with the multi-bank cache, providing slightly better results for a 4-bank cache. However algorithm 1 actually gets worse when the number of banks increases to 8. Furthermore, it fails to improve upon the multi-bank cache in the 2-bank case where a full copy cache is consid-

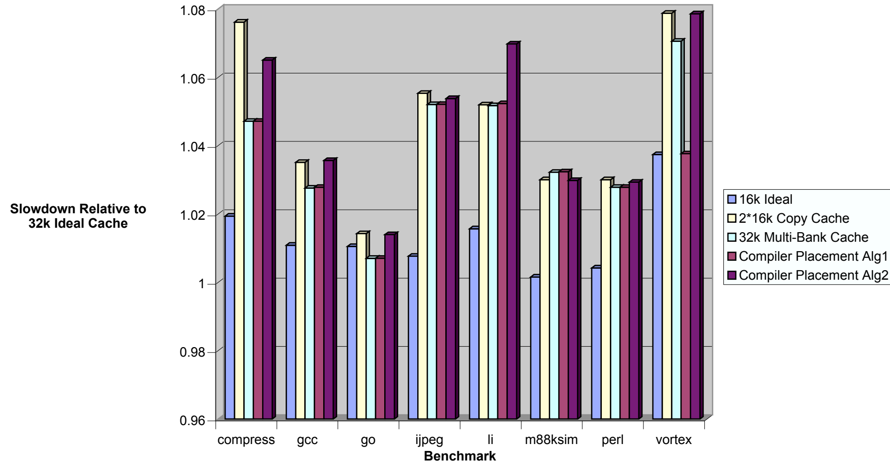


Figure 4: Comparison of various cache bank placement algorithms with 2 banks and 32k total cache space. All results are normalized against a 32k idea dual port cache.

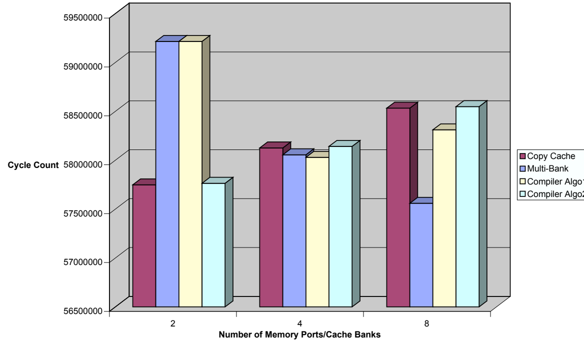
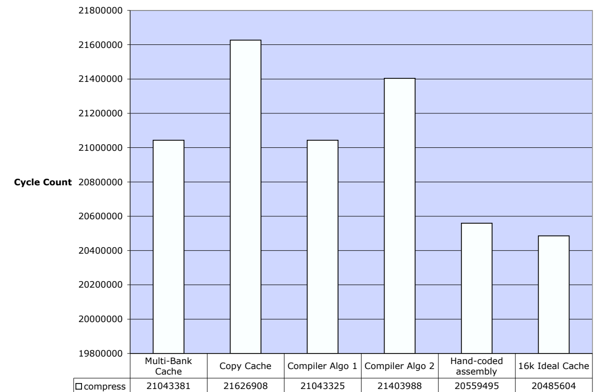


Figure 5: Comparison of various cache bank placement algorithms applied to the `linpack` benchmark as the number of cache banks is increased.



	Multi-Bank Cache	Copy Cache	Compiler Algo 1	Compiler Algo 2	Hand-coded assembly	16k Ideal Cache
compress	21043381	21626908	21043325	21403988	20559495	20485604

Figure 6: Demonstration of improvement possible if compiler technology can be improved.

erably faster.

Finally, in Figure 6 we demonstrate just how limited our current compiler infrastructure is. We hand-annotated the assembly for a single frequently called function in the compress95 benchmark. The compiler could not make the same annotation since the alias analysis failed to identify a certain pair of memory references as not aliasing. By just changing 16 lines of assembly code we improved the performance of the benchmark to be competitive with a 16k ideal dual ported cache.

6 Conclusion

Through the use of dynamic traces collected from the Simplescalar simulator, we have been able to investigate several aspects relevant to the static placement of data in a multi-banked cache. We primarily focused on two areas — the natural sets of memory instructions which access disjoint pieces of memory and contention between memory instructions. We found that keeping stack accesses from interfering with other memory instructions is relatively important for finding a large number of disjoint sets for bank assignment. We also discovered that aggressive inlining can aid in the creation of new disjoint sets of instructions. Inlining fails to be very effective, however, in getting rid of the problematic dominating set that appears to stand in the way of effective bank assignment.

The dominating set found in each of our disjoint instruction sets appears to reduce the performance increases possible with static cache placement, since

most contention occurs between instructions within the same set. In particular, by far most contention happens between the dominating set and itself, both statically and dynamically. As a result, it appears as if it should be nearly impossible to produce efficient bank assignments that reduce bank contention unless assignment to multiple banks or dynamic assignment support is provided.

Finally we evaluate the practicality of constructing efficient compiler algorithms for handling static assignment of cache banks to memory instructions. While some amount of careful inter-procedural full-program analysis would be essential for correctness, we have found that most performance gains should be able to be attained through less difficult intra-procedural analysis.

We also described compiler controlled split caches and our initial implementation of a compiler framework which is capable of optimizing for them. Although our results are mixed, they do indicate that there is definite potential for a cc-split cache's performance to exceed more traditional cache implementations requiring similar amounts of chip area. However, in many cases it appears that by accepting a marginal increase in chip area (doubling or quadrupling the number of banks, for example) a hardware solution can significantly improve in performance. In order to obtain a similar improvement in performance using cc-split caches it would be necessary to dramatically increase the complexity of the compiler and linker (to support inter-procedural analysis). Furthermore, since the compiler is explicitly managing the cache placement, it would be easy for the compiler to introduce subtle errors into programs which would be extremely difficult to track down. A cc-split cache could provide an effective way to increase memory parallelism without increasing cache complexity and chip area requirements, but is probably only suitable when increasing chip area usage even slightly is prohibitively expensive.

References

[1] Todd M. Austin and Gurindar S. Sohi. High-bandwidth address translation for multiple-issue processors. In *ISCA*, pages 158–167, 1996.

[2] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Memory bank disambiguation using modulo unrolling for raw machines, 1998.

[3] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308, 1996.

[4] B. Calder, K. Chandra, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, 1998.

[5] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, 1999.

[6] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999.

[7] Intel Corporation. *Pentium Processor User's Manual*. 1993.

[8] Zarka Cvetanovic and Dileep Bhandarkar. Performance characterization of the alpha 21164 microprocessor using TP and SPEC workloads. In *HPCA*, pages 270–280, 1996.

[9] Intel Corporation. *Intel Itanium 2 Processor Reference Manual*, 2002.

[10] Toni Juan, Juan J. Navarro, and Olivier Temam. Data caches for superscalar processors. In *International Conference on Supercomputing*, pages 60–67, 1997.

[11] R. E. Kessler, R. Jooss, A. Lebeck, and M. D. Hill. Inexpensive implementations of set-associativity. In *Proceedings of the 16th annual international symposium on Computer architecture*, pages 131–139. ACM Press, 1989.

[12] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., 1997.

[13] Wilfried Oed and O. Lange. On the effective bandwidth of interleaved memories in vector processor systems. *IEEE Transactions on Computers*, 34(10):949–957, 1985.

[14] Ram Raghavan and John P. Hayes. On randomly interleaved memories. In *Proceedings of the 1990 conference on Supercomputing*, pages 49–58. IEEE Computer Society Press, 1990.