Partitioned First-Level Cache Design for Clustered Microarchitectures

Paul Racunas
The University of Michigan
Ann Arbor, Michigan 48109-2122
racunas@eecs.umich.edu

Yale N. Patt
The University of Texas at Austin
Austin, Texas 78712-1084
patt@ece.utexas.edu

ABSTRACT

The high clock frequencies of modern superscalar processors make the wire delay incurred in moving data across the processor chip a significant concern. As frequencies continue to increase, it will become more difficult for a centralized first level data cache to supply the timely data bandwidth required by superscalar processors.

This paper presents a complete solution for the partitioning of the first level of the memory hierarchy. The first level data cache is split into several independent partitions, which are arbitrarily distributable across the processor die. After being decoded, memory instructions are sent to the reservation stations of the functional unit adjacent to the cache partition that they are most likely to access. The partition assignments for both static instructions and cache data are dynamically changed to adapt to data access patterns. A data cache line is permitted to reside in only one partition at a time, allowing each store to update only a single partition, and allowing the partitioning and simplification of the memory disambiguation logic. The partitioned cache achieves a reduction in cache access latency through a combination of reduced wire delay and reduced cache array size. A partitioned cache with eight 8KB direct-mapped partitions maintains a hit rate greater than that of a 32KB direct-mapped cache. A machine utilizing the partitioned cache outperforms a machine with a conventional 64KB direct-mapped cache by 4.5% and a machine with a 64KB 8-way set-associative cache by 7.0%, when cache latencies estimated through the use of the CACTI cache simulation tool are taken into account.

Categories and Subject Descriptors

C.1.1 [**Processor Architectures**]: Single Data Stream Architectures; B.3.m [**Memory Structure**]: Miscellaneous

ICS'03, June 23–26, 2003, San Francisco, California, USA.

General Terms

Performance, Design

Keywords

clustered microarchitecture, partitioned cache

1. INTRODUCTION

As clock speeds and microprocessor complexity continue to increase, additional pressure is placed on the memory system to supply more data in a timely fashion. This has led to a concerted research effort to reduce the latency and increase the bandwidth of the memory subsystem. Nowhere are the latency and bandwidth requirements more demanding than at the first level of the cache hierarchy. At a time when superscalar processors are beginning to segment functionality into separate clusters [8, 9, 7, 14], designing a centralized low latency first level cache is becoming more and more difficult.

This paper proposes the use of data partitioning to provide a distributable, scalable, fast-access, high-bandwidth, first-level caching structure. The partitioning framework consists of several independent cache arrays placed adjacent to their respective load units. These partitions communicate data with each other only through the second-level data cache. After passing through the decode and rename stages, memory instructions are sent to the reservation stations adjacent to the cache partition most likely to contain their data.

Initially, static loads are assigned to partitions on a round-robin basis. Each miss by a static load will fetch a cache line into its partition, and the data cache line will be assigned to the partition that requested it. As the static loads request their working set, the set of data used by each static load will either be gradually moved into the load's local partition, or the subsequent instances of the static load will be re-assigned to the partition in which their data resides. A set of two-bit counters injects enough hysteresis into the transitions to keep the instructions and data from repeatedly adjusting their assignment between partitions. A load balancing scheme monitors the number of misses to each partition, and changes the parameters of the partitioning algorithm to reassign instructions and data cache lines away from an individual partition with an abnormally high miss rate.

Employing the partitioning scheme results in several performance benefits. First and foremost is reduction in cache access latency due to both reduction in the wire delay

between the functional units and the cache array and reduction in the cache array access time itself. Secondly, the bandwidth of the first level caching structure is increased, both to the load units and to the second level data cache, since each partition can provide data and receive fills independently. Third, since most stores impact only the local partition, the memory disambiguation logic need only compare the locally pending loads and stores to detect memory dependencies. Fourth, the partitioned cache is entirely compatible with previously published methods for increasing effective set-associativity or cache bandwidth without explicitly implementing associativity or adding ports. Finally, a partitioned cache allows a larger effective cache to be accessed with the page offset bits of the virtual address, just as a set-associative cache does.

This paper shows that 64KB of storage can be arranged so that it can be accessed with the cache array latency of an 8KB direct-mapped cache. The storage can be arbitrarily distributed across the chip to minimize the wire delay between the cache array and each of the load/store functional units. A partitioned cache consisting of eight 8KB direct-mapped partitions yields a hit rate slightly better, on average, than that of a 32KB direct mapped cache. When realistic latencies and port limitations are considered, a single-cycle, partitioned cache with eight 8KB direct-mapped partitions and two read ports per partition outperforms a three-cycle, four-ported 64KB direct-mapped unpartitioned cache by 4.5%. This paper will also show that the partitioned cache is effective in increasing bandwidth and reduces the hardware complexity of the memory disambiguation logic.

2. RELATED WORK

There has been a great deal of research into increasing the hit rate and decreasing the access times of low-level caches. A number of schemes have been proposed to achieve an approximation of set-associativity with access times approaching that of a direct-mapped cache. The hash-rehash cache [1, 2] and way prediction [4] have been developed as methods for a load to potentially hit with a direct-mapped access regardless of where its target lies in the order of recently-used set elements. The hash-rehash cache uses a direct-mapped access followed by a second access using a different hashing function on a failure. Way prediction uses the instruction address of a load or the value of its source register to predict which element of the set is the desired one. Other techniques combine a direct-mapped data array with a set-associative tag array [19, 3], as for small associativities, an associative tag lookup is not significantly slower. The common theme of these methods is to combine a direct-mapped access with either another direct-mapped access or a partial [12, 11] or full set-associative tag match on failure, with preference to most-recently used elements. [16] These techniques, however, seek to optimize access to a single, monolithic structure. Partitioning strives to reduce access latency by the orthogonal methods of distributing the cache across the processor die and reducing the size of the cache array itself. Any of the above methods could be used in conjunction with a partitioned cache to further increase its associativity.

There has been less research in the microprocessor community examining dynamically splitting a cache into separate entities, as partitioning does. In [17], Wolfe proposes splitting the data cache in half and dedicating half to integer data and half to floating point data, allowing memory instructions of each type to be directed to the cache holding the proper type of data. Others have proposed splitting the data cache between the stack and the heap memory regions [6, 5]. While these partitioning schemes identify data streams that are very independent, they are only able to map data of a particular type to each half of their split cache. These methods are not scalable and can result in a suboptimal split, since stack caching requirements are very small and floating point data may or may not be present in a given application. In [18], Yoaz presents a dynamic bank prediction scheme for a microprocessor Each load unit is associated with a particular cache bank and is placed physically near it. Techniques from branch prediction and address prediction are used to predict which bank a memory instruction will access before it is steered to a functional unit. A bank misprediction results in the load being flushed from the machine and rescheduled. While two banks were used for all the results in this paper, Neefs [10] extends the notion to predict among a larger number of banks. Using an infinitely-sized hybrid of a stride and context-based bank predictor, he is able to map an instance of a load to the correct bank on average about 87% of the time, for a configuration with eight banks. In [13], Limaye and Shen propose a method of increasing L0 cache bandwidth by assigning static loads that are likely to access the data cache in the same cycle to separate cache subsections, duplicating cache lines as necessary. They are able to map a load to the correct cache subsection about 93% of the time. The bank prediction accuracies of these schemes are substantially lower than the accuracy of the method proposed in this paper.

3. MOTIVATION

The primary goal in implementing the first level of the memory hierarchy is to design a caching structure that minimizes latency and maximizes bandwidth with the minimum amount of hardware. With recent increases in clock frequency, wire delay has become a significant additional component of cache latency. In this environment, maintaining a centralized cache will be increasingly awkward. Due to wire delay, the Alpha 21264 was forced to implement a two-cycle first level data cache at a stated degradation of 4% in processor performance. [9] In this environment, a first level caching structure is needed that minimizes wire delay and maximizes bandwidth while maintaining a high hit rate.

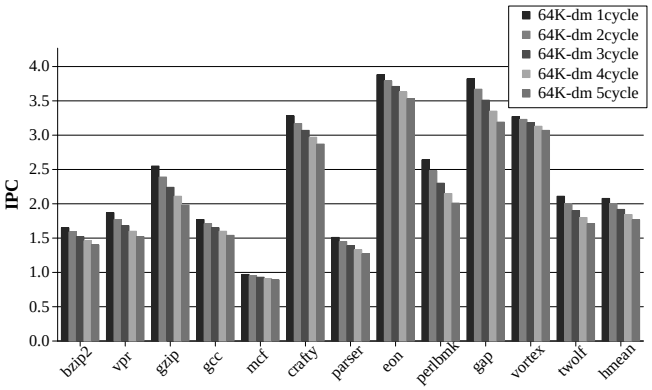

Figure 1: IPC vs. first-level cache latency

Figure 1 shows that, for our machine model, each cycle reduction in first level cache access latency results in a fairly constant linear IPC improvement. Starting with a five-cycle, 64KB direct-mapped first level cache (rightmost bar), and reducing its latency by one cycle at a time, the graph shows successive IPC improvements of 4.3%, 8.5%, 13.10%, and 17.61%. Clearly, a technique allowing a significant decrease in first-level cache access latency while maintaining the cache hit rate would be beneficial.

The solution proposed in this paper is to partition the first level of the memory hierarchy into several separate and independent entities, each physically located adjacent to one or more load/store functional units. After decoding a memory instruction, a prediction is made as to which partition is most likely to contain the data that the instruction will request. The memory instruction is then steered to the load/store unit associated with this partition. All communication between partitions occurs through the second level cache.

3.1 Partitioning Algorithm Overview

Static memory instructions are associated with a partition number, as is each line in the second level data cache. Load instructions trigger partitioning decisions only when a request misses in the first level of the cache hierarchy. Two degrees of freedom are available to the algorithm on a load that causes a partition miss. First, the line in the second level data cache can be assigned to the requesting partition. This may require invalidating the cache line in another partition first. Second, the assignment of the static load can be changed so that the next time it is seen it is steered to a different partition. The first option has more potentially harmful consequences. If a data cache line is moved to another partition, any other static loads that also need that data will now incur a miss when they next attempt to access it. On the other hand, if the memory instruction itself is re-assigned, it is merely steered to the new partition the next time it is decoded. This option creates a problem only if most of the data accessed by that particular memory instruction resides in the original partition. The implemented partitioning algorithm reflects a preference for partition reassignment of instructions over reassignment of data.

4. HARDWARE DESCRIPTION

The basic components necessary to partition the first level of the memory hierarchy can be seen in figure 2. The dotted lines in the figure designate the two partitions of the first level of the memory hierarchy. An instruction partition assignment table (PAT) tells the steering logic where to direct each memory instruction. A data partition assignment table (the global data PAT) is used by the second level of the memory hierarchy to ensure that a data cache line is present in only one partition at a time. Local copies of the global data PAT are used in each partition to allow memory disambiguation to be handled locally within the partition. Each entry in a partition assignment table holds a *partition identifier* and two-bit saturating *instruction hysteresis counter*. The partition identifier field of an instruction partition assignment table entry maps a memory instruction to the cache partition where its data is likely to be found. The data partition assignment table contains an entry for each line in the second level data cache. The partition identifier field of a data PAT entry maps each second level data cache line to the partition to which it it currently assigned.

After an instruction has been decoded and its dependencies have been mapped, it is issued to the reservation stations of the load/store functional unit corresponding to its partition identifier. Instructions being seen for the first time will have no partition identifier, and will be assigned to partitions in a round-robin fashion. Once the memory instruction arrives at its load/store unit, it waits in the reservation stations until all of its dependencies have been resolved.

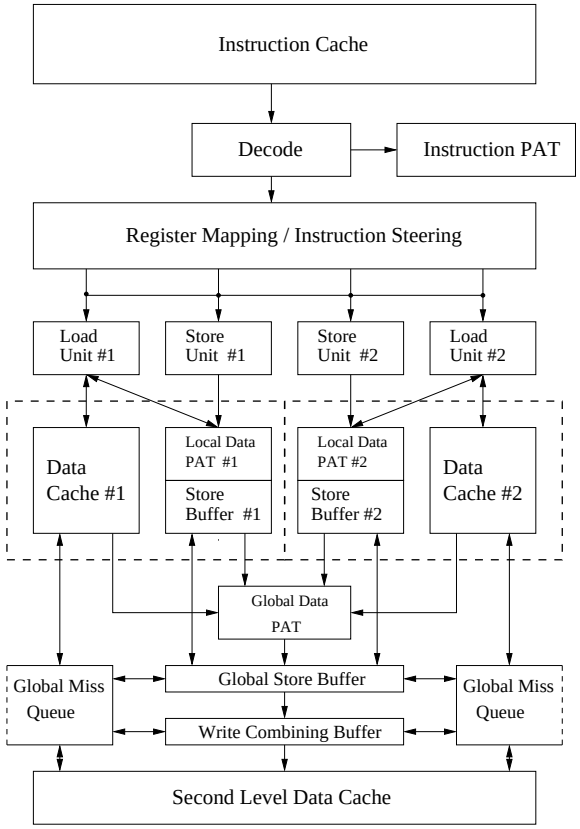

Figure 2: Organization of partitioned processor

4.1 Load Instructions

Once its dependencies have resolved, a load instruction is issued to its adjacent functional unit and accesses its cache partition. If the access hits in the partition, all is well. If the access is a miss, it is placed in the miss queue, and a request is made to the second level cache for the relevant cache line. Simultaneously, the miss request address is also sent to the global data PAT. If the request hits in the second level cache, it will have a corresponding entry in the global data PAT. The partition identifier listed in the global data PAT is then compared to the number of the partition that generated the miss request. A match in these two values implies that the data cache line had already been assigned to the requesting partition, but the line had been displaced by other data. Since this indicates that the assignment of cache line to partition had been accurate, the saturating hysteresis counter corresponding to the cache line in the global data PAT is incremented. If the two partition numbers do not

match, the corresponding hysteresis counter in the global data PAT is decremented. In this latter case, the partition miss is coming from a partition other than the one to which the cache line was assigned. A decision must now be made as to whether or not to reassign the cache line to the requesting partition by changing the partition identifier recorded in the global data PAT. This reassignment occurs only if the hysteresis counter had the value zero before the attempted decrement. If reassignment does not occur, a fill is still sent to the requesting partition. This fill provides the necessary data to any loads waiting for it in the miss queue, but the fill will not write any data into the cache partition itself. The latency of the load instruction that generated the miss request will be that of a second-level cache access.

The reassignment of a data cache line in the global data PAT requires several steps. First, the cache line must be invalidated in the previously assigned data cache partition. Any entries in the local store buffer of the old partition corresponding to this cache line must also be marked invalid. A placeholder is set in the local store buffer of the new partition that causes loads dependent on these entries to get their information from the global store buffer. Next, the new partition assignment information is sent to each local copy of the global data PAT. Lastly, the hysteresis counter in the global data PAT entry corresponding to the data cache line is set to 1, so that it is less likely to be immediately reassigned to another partition. The second level data cache is now permitted to send the fill to the requesting partition.

A static load's entry in the instruction PAT is updated at retire time. If the load instruction hit in the first level of the memory hierarchy, its partition identifier remains unchanged, and its instruction hysteresis counter is incremented. If the load access was a miss, the hysteresis counter is decremented. If the hysteresis counter contained the value zero before the attempted decrement, and the load instruction missed, the partition identifier of the load instruction is changed to the partition number that was recorded in the global data PAT at the time of the miss. Loads that miss in their partition but hit in the miss queue will wait there for the second level cache to provide the fill for the original outstanding miss request. They will receive the same partition assignment information as the original outstanding miss.

4.2 Store Instructions

The memory hierarchy used for the experiments in this paper processes store instructions slightly differently than a typical architecture might. Instead of writing store data into the first level cache with a write-through protocol to the second level cache, stores are written into a write combining buffer and sent directly to the second level cache. Once the second level cache has processed the store, it will issue a fill to the first level cache to update the cache line. The local and global store buffers are used to forward data to any load that requires the data in the interim. This technique is used for both partitioned and unpartitioned caches, as it allows over 65% of stores on average to be combined or eliminated.

When a store instruction is encountered, an entry is allocated for the store in both the local store buffer associated with its partition and in the global store buffer. Both these entries will be updated with the store's address and data once they have been calculated. Once the store's address has been calculated, the local copy of the global data PAT is accessed, to determine if the store data address is assigned to the local partition. If it is, store forwarding from the local store buffer is enabled for the allocated entry. If not, store forwarding from the entry in the local store buffer is disabled, and the store's information is propagated to the local store buffer of the relevant partition as soon as its data and address become known. Stores will also change the partition assignment information in the global data PAT, based on whether or not the cache line they update was assigned to the partition that they were steered to. While only load misses update the global data PAT, each store updates the fields of the structure. When the store retires, the store's data is sent to the write-combining buffer and the data waits there for an opportunity to update the second level cache.

A static store's entry in the instruction PAT is updated after it retires. If the access to the local data PAT showed that cache line associated with the store's address was assigned to the partition that the store was steered to, the store's instruction hysteresis counter is incremented. If the local data PAT showed that the store's data cache line was assigned to another partition, the store's instruction hysteresis counter is decremented. If a decrement is attempted with a hysteresis counter containing the value 0, the partition identifier associated with the static store is changed to the partition id from the local data PAT.

4.3 Memory Disambiguation

In most cases, memory disambiguation is performed locally. Loads are speculatively assumed to be dependent only on earlier stores that have been sent to the same partition. Within a partition, any memory dependence scheme can be used. There are two situations where cross-partition memory dependencies can occur. The first situation consists of a store steered to the wrong partition generating data to be used by a load steered to the correct partition. In this case, the load will speculatively execute. The store will identify that it was sent to the wrong partition when it accesses its local data PAT. It will propagate its address and data to the correct partition as soon as they are available. The dependent load will have to be re-executed. The second situation consists of a load steered to an incorrect partition being dependent on a store steered correctly. In this case, the load will miss in its local partition's data cache, and get the required data from the global store buffer. On all partitioned configurations tested with eight partitions or fewer, less than 2% of load forwards were cross-partition forwards or required an access to the global store buffer.

5. SIMULATION ENVIRONMENT

The experiments presented in this paper were performed on the SPEC2000 integer benchmark suite compiled for the Alpha EV6 ISA with -fast optimizations and profiling feedback enabled. The benchmarks are run to completion on the SPEC test inputs or a shortened version thereof. The baseline machine model is an aggressive 16-way superscalar machine with an idealized front-end, able to predict and fetch the targets of three branches every cycle. Table 1 shows a summary of the relevant machine parameters.

A perfect memory dependence predictor was used to delay dependent loads until their data was available. The L1 cache is locked during a fill, meaning that only loads dependent on fill data can execute while the write is progressing.

However, fills are scheduled to begin on cycles during which no loads are pending. Since the data bus between the L2 and L1 is 32 bytes, and the cache line size is 64 bytes, loads are not delayed for more than one cycle by the processing of a fill transaction.

Table 1: Machine Model

Branch Prediction
128K-entry gshare/PAs hybrid with 64K-entry hybrid selector; 4K-entry 8-way associative BTB, 32-entry call/return stack; 64K-entry indirect branch predictor. All predictors capable of 3 predictions / cycle;
Instruction Fetch
64KB, 4-way associative instruction cache with 3 cycle latency. Capable of three accesses per cycle.
Pipeline
4 cycle decode, 10 cycle register mapping 20 cycle overall misprediction penalty
Core
512-entry instruction window, 16 general purpose pipelined FUs;
Caches
3 cycle L1 data cache; 64B lines; 64-entry store buffer 8 L1 read / 1 L1 write ports; 32B full speed L1/L2 data bus; 512KB 8-way associative unified L2, 10 cycle latency; 64B lines; all intermediate queues and traffic are modeled.
Busses and Memory
16 outstanding misses; 32B memory bus at 2:1 bus ratio; split address/data busses; 1 cycle bus arbitration; 100 cycle DRAM access latency; 32 DRAM banks; all intermediate queues and traffic are modeled.

6. EXPERIMENTAL RESULTS

Traditionally, the benefit of a new caching scheme is measured by its effectiveness across structures of equivalent size. This method is useful for measuring relative cache storage efficiency. However, with processors shipping today with 512KB on-chip caches, cache size is clearly not the primary limitation for the first level of the memory hierarchy. The size of a first level cache is limited by its access time, not by available chip area. Hence, caching schemes for the first level of the memory hierarchy should be evaluated by performance across structures of equivalent access latency, rather than equivalent size. The default partitioned cache used in the following experiments consists of 64KB of storage broken into eight 8KB direct-mapped partitions. If its wire delay advantage is ignored, the partitioned cache has a cache array access time equivalent to that of an 8KB direct-mapped cache. Hence, the baseline for all of the speedup graphs presented in this section will be the performance achievable with a single, unpartitioned, 8KB, direct-mapped data cache. The performance of this baseline configuration exceeds that of a 64KB direct-mapped banked cache with 8 banks utilizing bank prediction as described in [10], if a full cache miss is taken on a bank misprediction.

6.1 Basic Configuration

All experiments use two-bit counters for the instruction hysteresis counter and the address hysteresis counter. Also, invalidation and partition assignments for the second level data cache are done on the granularity of an entire cache line. Experimentation showed that the additional flexibility of allowing disjoint portions of the same cache line to be valid in two partitions simultaneously was outweighed by the additional misses incurred to transfer the cache line when the whole was needed. To factor out the bandwidth advantage of the partitioned cache, the initial experiments approximate infinite bandwidth to the first level of the memory hierarchy by simulating eight read ports to each cache or partition. All cache configurations have a three-cycle access latency and eight read ports per cache or partition unless otherwise specified. All graphs except Figure 11 use a partitioned cache with an infinite instruction PAT and no additional latency for cross-partition memory dependencies.

Figure 3 shows the speedup of the basic partitioning scheme and various-sized unpartitioned caches over a processor with an unpartitioned 8KB direct-mapped cache. All caches are assumed to have an equivalent access latency of 3 cycles. The bar labeled "8-8KB-dm partitions" represents the performance of the basic partitioned cache with eight 8KB direct-mapped partitions. The other bars represent unpartitioned caches of various sizes and associativities. The graph shows that, even if access times were equal, the harmonic mean performance of the basic partitioned cache with 64KB of total storage broken into eight 8KB partitions is practically identical to that of a 32KB direct-mapped cache. The partitioned cache, containing 64KB of total storage, is able to outperform the 64KB direct-mapped cache on crafty, gap, and twolf. This is because these benchmarks are among those particularly sensitive to associativity. The partitioned cache is able to provide a small degree of associativity by assigning conflicting static loads to separate partitions. On the other hand, parser responds well to associativity, but sees little benefit from the partitioned cache. The problem in the case of parser is that the conflict misses are occurring between static instructions that partially share data. Hence, they must be mapped to the same partition, where they exhibit contention.

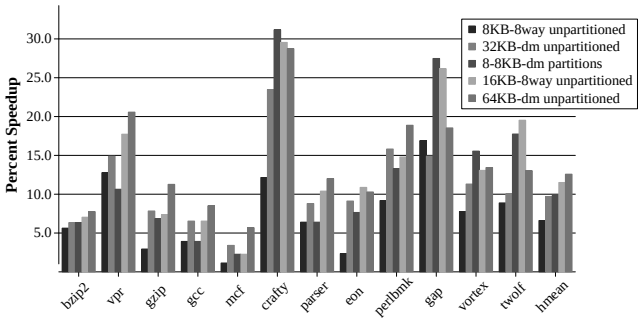

Figure 3: Basic Partitioning vs. Various Unpartitioned Configurations

Figure 4 shows how the performance of the partitioned cache changes as the number of partitions is varied. Ideally, the bars here would be almost equivalent. However, as the degree of partitioning increases, the size of the working set that can be kept in cache for any individual static load instruction decreases. Benchmarks that have individual loads with large working sets will perform better with fewer, larger partitions. These benchmarks exhibit the linear performance increase with decreasing partition degree that can be seen most noticeably in bzip2, gcc, mcf and parser. Crafty, gap and twolf respond primarily to associativity and often show an increase in performance with an increase in parti-

tion degree. In these benchmarks, the working set size that can be supported for a given static load instruction is less important than the ability to resolve conflict misses between static loads that do not share data. A partitioned cache with two partitions outperforms the basic eight partition cache by 2.6%, and one with four partitions outperforms the eight partition cache by 2.0%. A sixteen partition cache peforms on average, 1.9% worse than the eight partition cache. All partitioned caches in this graph are direct-mapped and have a three-cycle access latency.

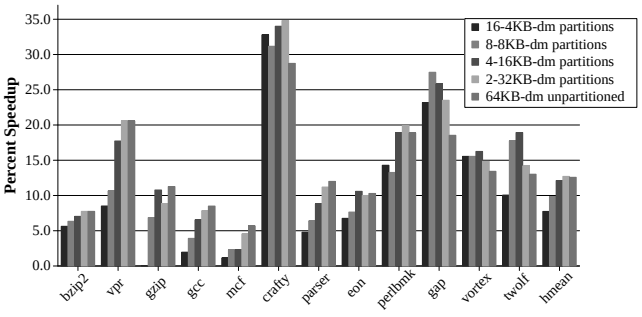

Figure 4: Comparative Performance of Different Degrees of Partitioning

6.2 Load Balancing

Since the basic partitioning scheme relies on the initial round-robin assignment of instructions to partitions to distribute cache lines, it is susceptible to worst case scenarios. For example, it has no mechanism to prevent all of the data and instructions from migrating to a single partition. While the hysteresis counters, particularly in the global data partition assignment table, are usually effective at preventing the data movement that would result in this occurrence, a mechanism should be in place to detect and redistribute loads should such a concentration occur. Hence, a load balancing mode was developed for the partitioning scheme. Load balancing mode is triggered when an undue amount of miss traffic is coming from a single partition. When in this mode, the hysteresis preventing data and instruction movement is eliminated for the problem partition, making it easy for its instructions and data to migrate to other partitions. Load balancing is monitored by maintaining a miss counter for each partition. At regular intervals, the counters are compared. Load balancing mode is triggered when the largest counter value is several times greater than the next largest. The experiments in this section will compare a configuration that never enters load balancing mode with one that is always in load balancing mode and with one that enters load balancing mode if the largest miss counter has a value three times greater than the second largest. For these experiments, the counters were compared every twenty thousand misses.

During load balancing mode, all hysteresis counters are ignored for the purposes of moving instructions or data out of the problem partition. In addition, no new data cache lines or load instructions can be assigned to the problem partition until load balancing mode terminates. Figure 5 shows the effect of load balancing on the percentage of loads that access each partition. The y-axis shows the percentage of the total number of dynamic loads that access each partition over the course of the benchmark. Each color represents the dynamic load traffic to an individual partition, sorted by load traffic from most to least, where black represents the partition with the most load traffic. There are three bars for each benchmark. From left to right, these represent: a configuration with no load balancing, a configuration where load balancing mode is triggered when the largest miss counter is three times greater than the second largest, and a configuration implementing continuous load balancing. By comparing the black region of the leftmost bar with the other two, one can see that load balancing is clearly effective in eliminating worst-case scenarios. A worst-case scenario consists of enough data and instructions migrating to the same partition that the capacity of that partition is overwhelmed. By examining the black bars for vpr, gcc, and eon, one can see that each have over 50% of their load traffic going to the same partition in the configuration without load balancing (leftmost of the three bars). However, the most frequently used partition receives only 25%, 16% and 37% respectively of the load traffic in the configuration with continuous load balancing (rightmost of the three bars.)

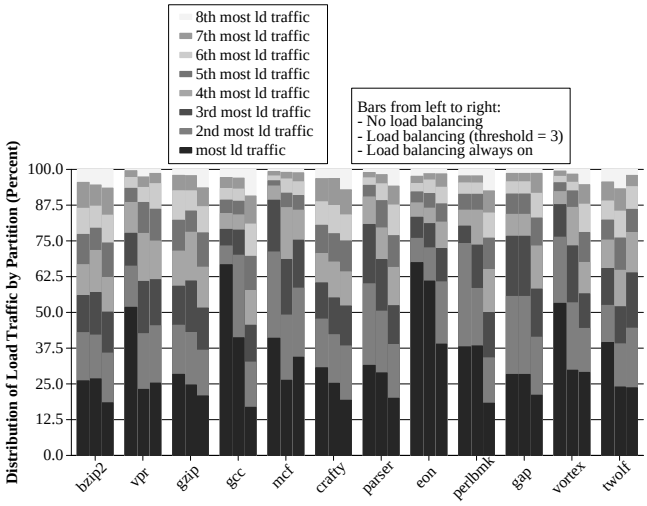

Figure 5: Effect of Load Balancing on Dynamic Load Distribution

Figure 6 shows how the frequency of misses in each partition corresponds to the amount of load traffic to that partition. Once load balancing is employed, only in bzip2, mcf, and twolf is the partition that receives the greatest fraction of dynamic loads also responsible for the greatest fraction of misses. With a load balancing threshold of 3 (middle bar), the partition receiving the sixth most load traffic in gzip (7.1% of the load traffic) is responsible for 41% of the misses. This illustrates one of the limitations of the partitioned cache. Since loads are assigned to partitions on a static load granularity, it is impossible to accommodate a static load with a footprint larger than the capacity of the partition. Similarly, without implementing set-associative partitions, it is impossible to eliminate conflict misses that occur between static loads that share data or between different instances of the same static load.

Employing load balancing involves a tradeoff, however. The goal of load balancing is to maximize hit rate by moving instructions with a large footprint to a partition that is

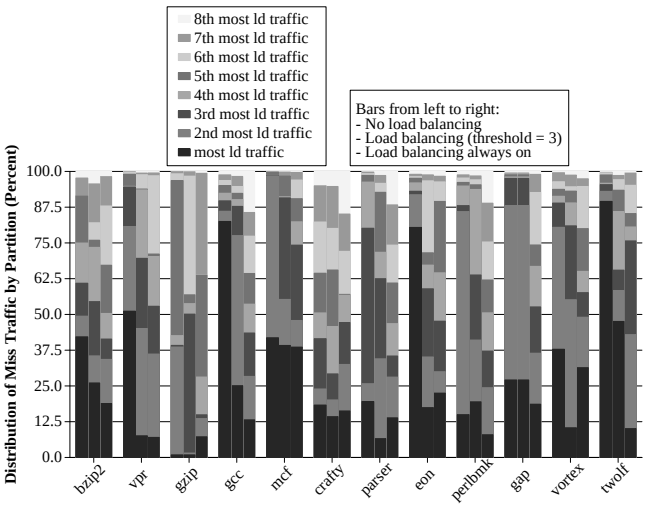

Figure 6: Effect of Load Balancing on Miss Distribution

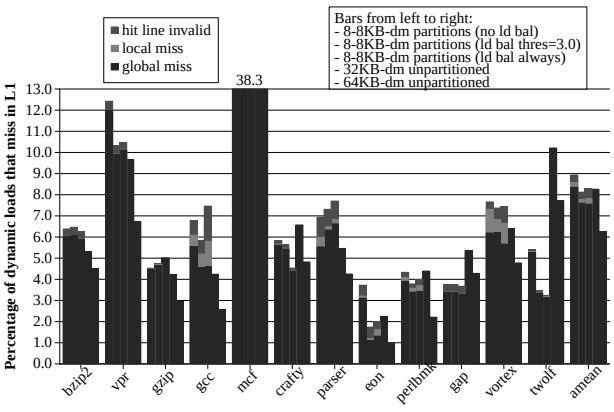

Figure 7: Overhead of Partitioning: Invalidation Misses

assigned mostly to loads with smaller footprints. However, each time a static load or data line is reassigned, there is the potential for triggering a number of misses from other loads that share that data. Also, if a single static load is responsible for producing the high miss rate, moving it to another partition will not do anything to solve the problem. With the continuous load balancing scheme, gcc shows a very even distribution of both load traffic and misses between partitions. (Compare the rightmost of the three bars for gcc in figures 5 and 6.) However, Figure 7 will show that this even distribution comes at the cost of a higher overall miss rate. The y-axis of this graph shows the percentage of dynamic loads that miss in their assigned partition or cache. These misses are broken into three categories. The term *global miss* represents a load access that misses in its assigned partition, and whose data is also not present in any other partition. The term *local miss* represents a load access that misses in the assigned partition, but whose data is currently present in a different partition. The term *hit line invalid* represents a load access that hits on the line in the local partition, except that the line is marked invalid. This is because it had been previously invalidated by the global data PAT as part of the process of assigning the cache line to another partition. While such "invalid" data may well be correct, and could be speculatively used by the load, this possibility is out of the scope of this paper. There are five bars for each benchmark. From left to right, these represent: a partitioned cache with eight 8KB direct-mapped partitions and no load balancing, the same partitioned cache with load balancing triggered when the largest miss counter exceeds the second largest by a factor of 3, the same partitioned configuration with continuous load balancing, a 32KB direct-mapped unpartitioned cache, and a 64KB direct-mapped unpartitioned cache. From the figure, we can see that while continuous load balancing produced an even distribution of loads and misses in gcc, it also causes a sharp increase in the number of reassignment misses (local misses and hits to invalid lines). In fact, gcc performs better with no load balancing at all than with continuous load balancing. These reassignment misses could potentially be eliminated by allowing data to reside in more than one partition at once. However, data duplication would reduce the available capacity of the partitioned cache as a whole in addition to increasing its complexity and damaging its ability to reduce the complexity of the memory disambiguation logic. Parser and gzip both show an increase in the global miss rate as the degree of load balancing is increased. This could be the result of temporarily moving several static loads with a large footprint into the same partition.

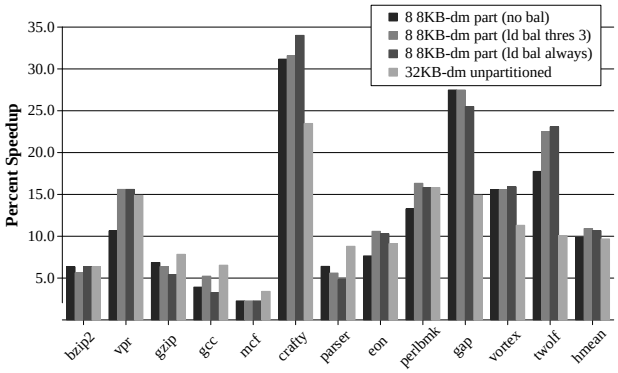

Figure 8: Effect of Load Balancing on IPC

Figure 8 shows the impact of the various load balancing schemes on IPC. As with all speedup graphs in this paper, the baseline of the graph is the performance achievable with an 8KB direct-mapped unpartitioned cache with 8 ports and a 3-cycle access time. When averaged across the benchmarks, a configuration with eight 8KB direct-mapped partitions with a load balancing threshold of 3 has an IPC about 1.1% greater than that of a machine with a 32KB direct-mapped cache of equal access latency. Since the load balancing scheme with a threshold of 3 provides the best performance, this load balancing method will be used for all data involving a partitioned cache from this point on.

6.3 Benefits and Limitations of the Partitioned Cache

There are several benefits to implementing a partitioned cache. In this section, most of these benefits are analyzed and quantified. The first benefit is the partitioned cache's reduction in cache access latency. The partitions

of the partitioned cache are arbitrarily distributable across the processor die. This minimizes the wire delay between the functional units and the cache array. This reduction in access latency due to this wire delay may well be more significant than the reductions in the cache array access latency that result from reducing the cache array size. The second benefit of the partitioned cache is the additional bandwidth inherently provided by the partitions. The partitioned cache increases bandwidth both between the first level of the cache hierarchy and the functional units and between the partitioned cache and the second level cache, since each partition acts independently. This section will show that the partitioned cache needs only two read ports per partition, while an unpartitioned cache requires at least three. Third, store handling and memory disambiguation logic are also partitioned. Memory disambiguation requires a large number of concurrent comparisons and complex hardware. Reducing the scope of comparisons required will significantly reduce this complexity. Fourth, the partitioned cache is compatible with previously published methods for increasing set-associativity or cache bandwidth without explicitly implementing associativity or adding ports. Lastly, the partitioned cache allows a larger effective cache to be accessed with the untranslated page offset bits of the virtual address, just as a set-associative cache does.

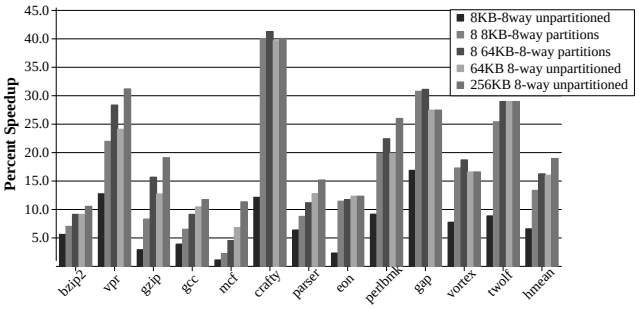

Figure 9: Performance of Partitioning with High Associativities and Large Caches

Figure 9 shows the response of the partitioned cache to increasing the associativity of each of the partitions. The y-axis of the graph shows speedup compared to the single, unpartitioned, 8KB direct-mapped cache baseline. A partitioned cache consisting of eight 8KB 8-way set-associative partitions achieves an IPC within 2.3% of that of a 64KB 8-way unpartitioned cache. However, the performance of the partitioned cache levels out more rapidly than the unpartitioned one, due to partition reassignment misses. A partitioned cache consisting of eight 64KB 8-way set-associative caches performs on average only slightly better than a single unpartitioned 64KB 8-way set-associative cache. The eight 8KB-8-way partitioned configuration outperforms the 64KB 8-way unpartitioned cache on gap by a significant amount. This is due to the partitioned cache's increased bandwidth to the second level cache. In gap, about 9% of the loads in the unpartitioned cache configuration are blocked for a cycle waiting for a store fill or L1 miss fill to be processed, while almost no loads are blocked by fills in the partitioned scheme. All configurations in this graph use the same 3-cycle access latency.

Figure 10 shows the performance effect of limiting the

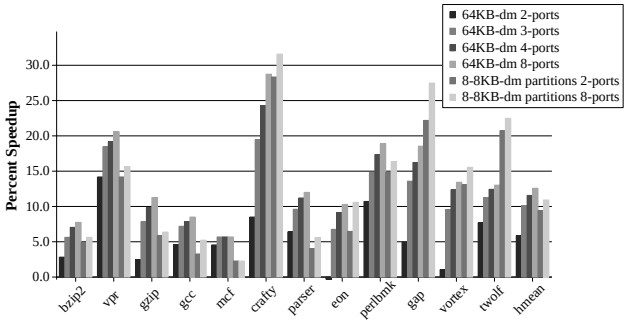

Figure 10: Bandwidth Analysis: Port Contention

bandwidth available to the first level memory hierarchy by reducing the number of read ports to the data cache. As always, the baseline used is an 8KB direct-mapped cache with 8 read ports. The graph shows that the partitioned cache succeeds in increasing the bandwidth to the data cache. The IPC dropoff between a partitioned cache with 8 read ports per partition and one with only 2 read ports per partition is on average 1.36%, comparable to the 0.91% dropoff going from the 64KB direct-mapped 8-ported unpartitioned cache to a 4-ported version thereof. When the 64KB direct-mapped unpartitioned cache is restricted to 2 read ports, it sees a 5.96% IPC dropoff from the 8-ported version. All configurations would see a large performance degradation if restricted to a single read port. This is due to the fact that there are regular periods of high traffic to certain regions of memory, such as the stack, that cannot be satisfied by a single port, even with partitioning.

While the partitioned cache's reduction in cache access latency due to wire delay is highly dependent on implementation, there are tools available to estimate its reduction in cache array access latency. Table 2 shows cache array timing estimates for caches with 64 byte lines under 0.13 micron technology. Cacti 2.0 [15] was used to generate these estimates. The table shows that a partitioned cache consisting of eight 8KB two-ported direct-mapped partitions has an array access time of 0.625ns in 0.13 micron technology. This is essentially half the access time of an equivalently-sized direct-mapped unpartitioned cache array using either 3 or 4 ports, more than 3 times less than a 64KB 8-way set-associative unpartitioned cache array with 3 ports, and 4 times less than a 64KB 8-way set associative cache array with 4 read ports.

Table 2: Cacti 2.0 Timing Data

Cache size	Ports	direct-mapped	8-way set-assoc.
8KB	1 rd, 1 rd/wr	0.625 ns	1.154 ns
8KB	3 rd, 1 rd/wr	0.734 ns	1.640 ns
16KB	1 rd, 1 rd/wr	0.713 ns	1.228 ns
16KB	3 rd, 1 rd/wr	0.875 ns	1.787 ns
32KB	1 rd, 1 rd/wr	0.869 ns	1.362 ns
32KB	3 rd, 1 rd/wr	1.088 ns	2.040 ns
64KB	1 rd, 1 rd/wr	1.057 ns	1.640 ns
64KB	2 rd, 1 rd/wr	1.211 ns	2.067 ns
64KB	3 rd, 1 rd/wr	1.378 ns	2.533 ns

It is important to note that the data provided by CACTI does not take into account the reduction in wire delay leading up to the cache array itself. This wire delay caused the Alpha 21264 to double the latency of its first level

cache access [9], and wire delay is expected to become an increasingly important problem as clock rates increase. The partitioned cache provides the ability to arbitrarily place the partitions and associated memory disambiguation logic anywhere on the processor floorplan. Hence, each partition can be placed immediately adjacent to the load and store units that access it. This paper will conservatively assume that this reduction in wire delay is able to reduce cache access time by at least 0.625 ns, the latency of an 8KB direct-mapped cache array access.

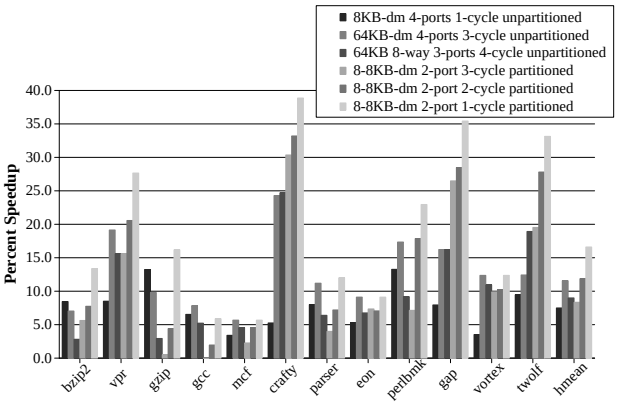

Figure 11: Potential Benefits of Optimized Cache Access Latency

Figure 11 incorporates all costs and advantages of the partitioned cache. A 20KB Instruction PAT and 5KB Global Data PAT are simulated. In addition, each cross-partition memory forward incurs a penalty of 10 cycles for the partitioned cache. No additional penalty is simulated for memory forwarding in the unpartitioned configurations. Together, the imposition of a real PAT and a 10-cycle cross-partition forwarding penalty reduce the performance of a single-cycle 8-partition partitioned cache by an average of 1%. This figure shows the performance achievable if we allow the true strength of the partitioned cache, the reduction in cache access latency, to take effect. The graph lists the access time of a 64KB direct-mapped unpartitioned cache with 4 read ports as 3 cycles (twice the latency of a 8KB direct-mapped 2-ported partition + 0.625 ns for wire delay.) Similarly, the access time of a 64KB 8-way set associative cache with 3 read ports is computed to be 3 cycles (twice the latency of a 8KB direct-mapped 2-ported partition + 0.625 ns for wire delay.) Performance for a partitioned cache with eight 8KB direct-mapped partitions and two read ports is shown for an access time varying from 3 cycles to a single cycle. Based on the CACTI data shown in table 2, a single-cycle cache access latency is the most reasonable estimate. The single-cycle partitioned cache outperforms the "64KB-dm 4-ports 3-cycle unpartitioned" configuration by 4.5%, and the "64KB-8way 3-ports 4-cycle unpartitioned" configuration by 7.0%.

7. CONCLUSIONS

A high speed, high bandwidth first level cache is critical to the performance of a superscalar processor. As clock frequencies continue to increase, it will become more and more difficult to achieve this requirement with a centralized structure. Already, current processors are including pipeline stages that do nothing but move data from one point on the chip to another. This paper presents a mechanism that allows the first level of the memory hierarchy to be arbitrarily distributed across the processor die, allowing wire delay to be minimized. The partitioning scheme functions without the need for compiler support or profiling information and will dynamically adapt to the data usage patterns of the running program. It can be combined with any previously published scheme for streamlining the data cache access or implementing associativity in a direct-mapped cache array. This paper has shown that further increasing the associativity or decreasing the access time of the partitioned cache through these methods will yield additional performance gains.

The partitioned cache has been shown to decrease cache access time and increase cache bandwidth while maintaining a reasonable hit rate. These benefits are achieved with a minimum amount of additional maintenance information, and the partitioned cache actually ends up considerably simplifying the hardware involved in memory disambiguation. A direct-mapped partitioned cache with eight 8KB partitions achieves a hit rate greater than that of an unpartitioned 32KB direct-mapped cache. When incorporating the likely reduction in cache access latency, a single-cycle two-ported partitioned cache consisting of eight 8KB direct-mapped partitions outperforms a 3-cycle 64KB direct-mapped by 4.5%, and a 4-cycle 3-ported 64KB 8-way set-associative cache by 7.0%. The access latency of the partitioned cache is equivalent to that of an 8KB direct mapped unpartitioned cache placed immediately adjacent to all load/store units that must access it.

The partitioned cache does have some weaknesses. Individual static loads that access data footprints larger than a single partition are awkward for the partition cache to handle, since it does not assign partitions on a finer granularity than instruction address. The partitioned cache, as presented here, cannot provide associativity between static loads that share the same data. A possible solution to this problem is to implement a different hashing function for each partition, in the hope that the loads will migrate to a partition where they will conflict less. The performance of large, high-associativity partitioned caches is dominated by invalidation misses, and does not compare favorably with equivalent unpartitioned caches. Future research will examine these issues.

8. ACKNOWLEDGMENTS

We gratefully acknowledge the Cockrell Foundation and Intel Corporation for supporting the research that led to this paper. We thank the anonymous reviewers and Eric Sprangle for their comments on earlier drafts of this work, as well as the other members of the HPS research group and associated research scientists for their continual interaction and insights.

9. REFERENCES

[1] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating systems and multiprogramming. *ACM Transactions on Computer Systems*, 6(4):393–431, Nov. 1988.

[2] A. Agarwal and S. Pudar. Column-associative caches: A technique for reducing the miss rate of

direct-mapped caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 179–190, 1993.

[3] B. Batson and T. N. Vijaykumar. Reactive associative caches. In *Proceedings of the 2001 International Conference on Parallel Architecture and Compilation*, 2001.

[4] B. Calder and D. Grunwald. Next cache line and set prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 287–296, 1995.

[5] S. Cho, P. C. Yew, and G. Lee. Access region locality for high-bandwidth processor memory system design. In *Proceedings of the 32th Annual ACM/IEEE International Symposium on Microarchitecture*, 1999.

[6] S. Cho, P. C. Yew, and G. Lee. Decoupling local variable accesses in a wide-issue superscalar processor. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 100–110, 1999.

[7] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 149–159, Dec. 1997.

[8] M. Franklin. The multiscalar architecture. Technical Report 1196, Computer Sciences Department, University of Wisconsin - Madison, Nov. 1993.

[9] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, pages 11–16, Oct. 1996.

[10] H. V. Henk Neefs and K. D. Bosschere. A technique for high bandwidth and deterministic low latency load/store accesses to multiple cache banks. In *Proceedings of the Sixth IEEE International Symposium on High Performance Computer Architecture*, pages 313–324, 2000.

[11] T. Juan, T. Lang, and J. J. Navarro. The difference-bit cache. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 114–120, 1996.

[12] R. Kessler, R. Joss, A. Lebeck, and M. Hill. Inexpensive implementations of set-associativity. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 131–139, 1989.

[13] D. Limaye, R. Rakvic, and J. P. Shen. Parallel cachelets. In *International Conference on Computer Design*, pages 284–292, 2001.

[14] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[15] G. Reinman and N. P. Jouppi. Cacti 2.0: An integrated cache timing and power model. Technical report, Western Research Laboratory, 2000.

[16] K. So and R. N. Rechtschaffen. Cache operations by MRU change. *IEEE Transactions on Computers*, 37(6):700–709, June 1988.

[17] A. Wolfe and R. Boleyn. Two-ported cache alternatives for superscalar processors. In *Proceedings of the 26th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 41–48, 1993.

[18] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation techniques for improving load related instruction scheduling. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

[19] C. Zhang, X. Zhang, and Y. Yan. Two fast and high-associativity cache schemes. *IEEE Micro*, pages 40–49, Sept. 1997.