

RICE UNIVERSITY

**Dynamically Reconfigurable Data Caches in Low
Power Computing**

by

Michael C. Brogioli

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Keith D. Cooper, Professor
Computer Science

J. Robert Jump, Professor
Electrical and Computer Engineering

Scott Rixner, Assistant Professor
Electrical and Computer Engineering
and Computer Science

Houston, Texas

August, 2002

Dynamically Reconfigurable Data Caches in Low Power Computing

Michael C. Brogioli

Abstract

In order to curb microprocessor power consumption, we propose an L1 data cache which can be reconfigured dynamically at runtime according to the cache requirements of a given application. A two phase approach is used involving both compile time information, and the runtime monitoring of program performance. The compiler predicts L1 data cache requirements of loop nests in the input program, and instructs the hardware on how much L1 data cache to enable during a loop nest's execution. For regions of the program not analyzable at compile time, the hardware itself monitors program performance and reconfigures the L1 data cache so as to maintain cache performance while minimizing cache power consumption. In addition to this, we provide a study of data reuses inside loop nests of the SPEC CPU2000 and Mediabench benchmarks. The sensitivity of data reuses to L1 data cache associativity is analyzed to illustrate the potential power savings a reconfigurable L1 data cache can achieve.

Acknowledgements

I would first off like to thank Scott Rixner and Keith Cooper for their insight and support on this project. I would also like to thank my third committee member, J. Robert Jump, for additional comments and criticisms which helped shape this document. Bryan Jones also contributed immensely during the early stages of this work. Finally, I would like to thank my family and Courtney who always took an interest in what I was working on. Without their support, none of this would have been possible.

Contents

Abstract	i
List of Illustrations	v
List of Tables	vii
1 Introduction	1
2 Background and Literature Review	3
2.1 Hardware Support For Reducing Power Consumption	3
2.2 Compiler Support for Predicting Data Cache Performance	5
2.3 Contributions	7
3 Motivational Example	9
4 Architecture	13
4.1 Reconfiguring the Cache via Hardware Performance Monitoring . . .	14
4.2 Reconfiguring the Cache via Compile Time Information	15
4.3 Enabling and Disabling Cache Ways	17
5 Compiler	20
5.1 Analyzable Loop Nests	20
5.2 Cache Miss Equations: What They Represent	22
5.2.1 Terminology	23
5.3 Generating Cache Miss Equations	25
5.3.1 Example: Forming CMEs	26

5.4	Solving the Cache Miss Equations	26
5.5	Cache Miss Equations to Cache Configuration	27
6	Experimental Setup	31
6.1	Simulated Architecture	31
6.2	Tool Chain	31
7	Results	35
7.1	Simulation Details	35
7.2	Baseline Cache Performance	36
7.3	Reconfiguring the Cache Via Hardware Performance Monitoring . . .	42
7.4	Reconfiguring the Cache Via Compiler Time Analysis of Loop Nests .	49
8	A Study of Loop Nest Data Reuses for SPEC CPU2000 and Mediabench Programs	53
8.1	Loop Nest Cache Analysis Framework	53
8.2	Loop Nest Cache Analysis Algorithm	54
8.2.1	Caveats to Loop Nest Reuse Analysis	56
8.3	SPEC CPU2000 and Mediabench Loop Nest Data Reuse Results . . .	57
9	Conclusions and Future Work	67
	Bibliography	69

Illustrations

3.1	A Simple Function With Varying Data Cache Behavior	10
4.1	SRAM with an NMOS Gated-Vdd	19
5.1	Simple Matrix Multiply Loop Nest	24
6.1	Block Diagram of Tool Chain	34
7.1	SPEC CPU2000 Fixed L1 Data Cache Size, L1 data cache miss rates	38
7.2	Mediabench Fixed L1 Data Cache Size, L1 data cache miss rates . . .	39
7.3	SPEC CPU2000 Fixed L1 Data Cache Size, Unified L2 cache miss rates	40
7.4	Mediabench Fixed L1 Data Cache Size, Unified L2 cache miss rates .	41
8.1	Algorithm to Analyze Loop Nest Data Reuses	55
8.2	Distribution of Reuses Realized Per Loops Nest vs. L1 Data Cache Associativity, SPEC CPU2000 Integer Programs, 10%–100%	59
8.3	Distribution of Reuses Realized Per Loops Nest vs. L1 Data Cache Associativity, SPEC CPU2000 Floating Point, 10%–100%	60
8.4	Distribution of Reuses Realized Per Loops Nest vs. L1 Data Cache Associativity, Mediabench Programs, 10%–100%	61
8.5	Distribution of Reuses Realized Per Loops Nest vs. L1 Data Cache Associativity, SPEC CPU2000 Integer Programs, 90%–100%	62

8.6	Distribution of Reuses Realized Per Loops Nest vs. L1 Data Cache Associativity, SPEC CPU2000 Floating Point Programs, 90%–100% . . .	63
8.7	Distribution of Reuses Realized Per Loops Nest vs. L1 Data Cache Associativity, Mediabench Programs, 90%–100%	64

Tables

6.1	SimpleScalar Simulation Parameters	32
7.1	Reconfigurable L1 Data Cache via Hardware Sampling, L1 and L2 Cache Performance	43
7.2	Reconfigurable L1 Data Cache via Hardware Sampling: Instruction Counts and Reconfiguration Statistics	44
7.3	Reconfigurable L1 Data Cache via Hardware Sampling: Relative Change in L1 Data Cache Miss Rates	45
7.4	Analyzable Loop Nests in SPEC CPU2000 and Mediabench Benchmarks	50

Chapter 1

Introduction

In recent years there has been a marked increase in microprocessor power consumption. Modern microprocessors can now have an average power dissipation of more than 70 Watts, as is the case with the Alpha 21264 [12]. This is due to increasingly complex hardware designs, increasing on-chip transistor counts, and increased clock rates. Many modern microprocessor designs also dedicate the majority of their transistors to on-chip instruction and data caches. As an example, 60% of the StrongARM's on-chip transistors are devoted to caches and memory structures [16]. Because of the large number of transistors which make up on-chip caches, they often account for a large portion of the total power consumed by modern microprocessors.

If microprocessor power consumption is to be reduced via architectural innovations or compiler technology, the on-chip cache is a good place to focus our efforts. In addition to consuming a significant portion of on-chip power, on-chip cache performance can vary within and across applications. Modern cache hierarchies are designed to satisfy demands of the most memory intensive application phases, but this is not the case for all phases of every application [19]. We propose an L1 data cache with a set associativity that can be adjusted according to the L1 data cache requirements of a given application phase. By doing this, we enable only enough L1 data cache associativity to maintain program performance. By not using the full set associativity at all times, parallel look-ups in the disabled L1 data cache ways can be avoided, and leakage current can be reduced.

In order to reconfigure the L1 data cache according to an applications L1 data cache requirements, we use both compile time information and runtime performance

monitoring. Our compiler builds a system of equations modeling the L1 data cache behavior of loop nests. Hints are then inserted into program executable which instruct the hardware on the amount of L1 data cache to enable during a loop nests execution. For regions of the program not analyzable at compile time, the hardware monitors program performance and adapts to the L1 data cache needs at runtime.

For a number of the SPEC CPU2000 and Mediabench benchmarks we show that it is possible to disable portions of the L1 data cache at runtime using only hardware performance monitoring. By doing this, significant power can be conserved in the L1 data cache while not decreasing overall program performance. Additionally, we show that it is difficult for the compiler to predict L1 data cache performance of loop nests in real programs. Although accurate models can be built for many loops in the programs analyzed, the amount of compile time information needed by our algorithm prevents compiler analysis of loops which account for the majority of program runtime. Finally, we show that a significant number of loops in both SPEC CPU2000 and Mediabench programs contain data reuses which are highly sensitive to L1 data cache associativity. This effectively shows that improvements can be made over adaptive mechanisms such as hardware performance monitoring if the compiler can analyze such regions of code.

Chapter 2

Background and Literature Review

The concept of reducing processor power consumption via a reconfigurable hardware mechanism is not an idea unique to this body of work. Various technologies have been developed to reconfigure processor clock speed, perform voltage scaling over time, and even enable and disable functional units. Additionally, the notion of predicting memory behavior of an application at compile time has been around for many years. Techniques such as software controlled cache prefetching, value reuse profiling, and even loop transformations all predict memory hierarchy performance at compile time. The following subsections describe in further detail both architectural innovations in reducing processor power consumption, as well as methods for predicting data cache performance at compile time.

2.1 Hardware Support For Reducing Power Consumption

In recent years, there has been an emergence of technologies in which the hardware can be reconfigured at runtime in order to reduce processor power consumption. Balasubramanian et al. [2] propose a system in which the hardware monitors an application's intolerance to cache misses, and reacts accordingly to improve memory hierarchy performance while taking energy consumption into consideration. The authors' proposed cache operates as a virtual two-level, physical one-level, non-inclusive cache hierarchy. The corresponding size, associativity, and latency of each of the two levels is chosen at runtime. By dynamically swapping cache lines between conventional L1 and L2 structures, hardware can be tuned to an application's behavior in order to maximize performance while minimizing power consumption. When applied to a two

level cache and TLB hierarchy, the authors' report an average 15% reduction in cycles per instruction versus a statically sized two level cache and TLB hierarchy. This corresponds to an average 27% reduction in memory cycles per instruction. By reducing the average number of cycles per instruction, total program runtime decreases. This, in turn, reduces the total power consumed during program execution.

In addition to reconfiguring hardware at runtime, predicting cache-way accesses on data and instruction cache lookups has recently been proposed as a means of reducing total cache power consumption [13] [20]. In a sequential access cache, as is the case with the Alpha 21164's L2 data cache [8], the cache waits until the tag array determines the matching way, and then accesses only the resulting way of the data array. This dissipates about 75% less power than a parallel access cache, but due to the serialized nature of the tag and data arrays lookups, it adds as much as 60% to the cache access time. The key to achieving parallel access cache performance with the power consumption benefits of a serial access cache lies in predicting the cache-way in which a data element resides, and only accessing the resulting cache-way [28].

Powell et al. [20] go one step further in combining way-prediction with a technique called selective direct-mapping. The performance benefits achieved via selective direct-mapping lie in the fact that on the order of 70% to 80% of L1 data cache accesses are non-conflicting, in that they do not map to the same set as another access. By determining at runtime which memory accesses do not compete with other references for a given cache line, cache-way prediction mechanisms can be bypassed. The memory address of the selectively direct-mapped reference explicitly maps to the cache-way containing the data. Powell et al. apply both selective direct-mapping and cache-way prediction to the L1 data cache, while only cache-way prediction is applied to the L1 instruction cache. Their results show an average L1 data cache energy delay reduction of 69% over a 4-way parallel access L1 data cache. For the L1 instruction cache, they report an average energy delay reduction of 64% over a 4-way parallel access instruction cache.

2.2 Compiler Support for Predicting Data Cache Performance

The prediction of data cache performance at compile time is necessary if our compiler is to optimize the L1 data cache configuration for power consumption. Ghosh et al. [10] propose a system of *Cache Miss Equations* which are generated and solved for at compile time. The Cache Miss Equations encapsulate the data reference stream of a loop nest, as well as the cold, capacity, and conflict misses that occur during the loop nest’s execution. A system of equations is constructed at compile time representing the potential L1 data cache misses that occur in the loop nest. Using the iteration space of the loop nest, and the reuse vectors of each array reference in the loop nest, the equations are solved to determine the exact L1 data cache misses that occur in each loop nest.

Cascaval et al. [7] propose a compile time model, which when combined with light weight profiling data, achieves an L1 data cache hit rate accuracy within 20% of the measured performance for codes using both dense and sparse computation methods. Implemented in the Polaris source to source translator [3], their strategy generates symbolic expressions for each loop of the program being analyzed. The resulting symbolic expressions incorporate predictions of CPU execution time and cache hierarchy behavior. In the case that compile time information cannot be obtained, either run time profiling or simple approximations can provide the necessary information. Additionally, in order to incorporate side effects of the input data set the compiler places very light instrumentation into the program. After the instrumented code is run, information on the input data is collected and substituted into the symbolic expressions.

In order to predict memory hierarchy performance, the authors of [7] proposed the model represented by Equation 2.1, in which T_{Mem} represents the time spent accessing memory locations, $CycleTime$ represents the processor clock cycle time, M_i represents the number of accesses that miss in the i^{th} level of the memory hierarchy, and C_i represents the penalty (in machine cycles) for a miss in the i^{th} level of the

memory hierarchy:

$$T_{Mem} = CycleTime * \sum_i^{numLevels} (M_i * C_i) \quad (2.1)$$

In order to estimate the actual number of cache misses that occur, both a stack distance model and an indirect access model are used, depending on how much compile time information is available. The stack distance model requires both accurate data dependence information and a program trace, and generates a stack histogram which illustrates the distribution of the number of references at different stack distances. In order to determine the number of cache misses M_i for a given cache size S_i , the compiler labels each data dependence arc with the number of distinct memory access locations referenced between the source and sink of the dependence. The dependence with the minimum number of intermittent memory references is assigned a stack distance of δ for the sink of the data dependence arc, and the number of times a static reference is accessed is A_δ . The resulting number of cache misses for the cache at level i with cache size S_i is computing as is shown in Equation 2.2:

$$M_i = \sum_{\delta=S_i}^{\infty} A_\delta \quad (2.2)$$

In the case that data dependence vectors cannot be determined at compile time, approximations on the number of cache lines spanned by an array reference must be made. Cascaval et al. propose obtaining the number of cache misses by multiplying each array reference A_j with the array element size e_j , and dividing the sum by the number of bytes for each block size in the memory hierarchy $BlockSize_i$. The number of misses can thus be expresses as is shown in Equation 2.3:

$$M_i = (\sum_j A_j * e_j) / BlockSize_i \quad (2.3)$$

In an analysis of I/O independent loops contained in the SPECfp95 benchmarks [22], Cascaval et al. reported cache miss prediction accuracy within 25% for many

of the loops. Certain levels of inaccuracy exist due to the fact that their model does not account for conflict misses. Similarly, execution time predictions for many of the loops remained within 25% as well. Results for highly optimized codes were somewhat worse due to inner loop reuse, which their in direct access model failed to capture.

In [18], Porterfield proposes another method for compile time cache performance prediction when precise data dependence information is available for loop nests. Building upon the notion that each iteration of a loop nest causes approximately the same number of unique memory blocks to be accessed between endpoints of a dependence, the *Overflow Iteration* is determined. The Overflow Iteration is the maximum number of iterations of a given loop nest which can have all of their data accessed by the loop maintained in cache at the same time. Any dependence edge in the loop which requires more iterations than the Overflow Iteration in order for its endpoints to remain in cache will result in a miss. The reasoning for this is that for a reuse to occur between the endpoints of the dependence edge, more unique memory blocks will need to be accessed than are available in the cache. Porterfield uses this analysis to determine the benefits of various reordering transformations which attempt to maximize the number of loop dependences with distances less than the overflow iteration. Additionally, the Overflow Iteration is used in software prefetching, as many of the cases in which hardware prefetching failed to achieve a performance increase pertained to loop array access patterns which could be detected statically at compile time.

2.3 Contributions

Rather than swapping lines between L1 and L2 data cache, or performing way prediction in attempts to reduce data cache power consumption, we propose a technique that combines a reconfigurable L1 data cache with compile time analysis of loop nest L1 data cache usage. The various ways in our set associative L1 data cache can be dynamically disabled at runtime in order to reduce leakage current and avoid costly

parallel cache way look-ups. The compiler analyzes loop nests in the input program, and determines the minimal amount of L1 data cache associativity necessary to avoid the maximum number of potential L1 data cache misses. Hints are then inserted into the program executable that instruct the hardware on the number of L1 data cache ways to enable during the loop nest's execution. For regions of the program not analyzable at compile time, the hardware itself monitors L1 and L2 cache performance and reconfigures the L1 data cache dynamically. By doing this, we effectively reduce the amount of power consumed by the L1 data cache while maintaining program performance.

Chapter 3

Motivational Example

Traditionally, first level data caches are sized to maximize performance subject to area and delay constraints. However, individual applications have different working set sizes and varying sensitivity to cache conflicts. The optimal cache size is therefore different for each application. It can also vary significantly over the course of execution of an application. By sizing the cache for the average case, microprocessors are able to achieve good performance across a wide range of applications. However, applications, or sections of applications, that cannot fully utilize the first level data cache pay a price, in terms of power dissipation, without receiving the comensurate benefit in performance.

To illustrate this problem, consider the data cache behavior of the code in Figure 3.1. For this example, we assume that the order in which the arrays are listed in the code is the order in which they are allocated by the compiler. In addition, no inter-array padding is used, and all data elements are assumed to be four bytes in size. This function contains three simple loops that cannot fully take advantage of a 64 KB 4-way set associative data cache. The first loop, Loop A, iterates over two 32 KB arrays of integers and performs a sum of products. Naively, this requires 64 KB of data, and could use the entire data cache. Obviously, this is unnecessary as there is no temporal reuse of data. Given that these arrays are allocated consecutively, they will be exactly 32 KB apart, so for every i , references to *source1*[i] and *source2*[i] will map to the same cache set (in this simple example, the compiler could offset the arrays so that this would not happen, but this is not always possible). Therefore, if only a single cache way were enabled, every memory reference would

```

void example() {
    int i, j, sum;
    int result[4096];
    int a[4096], b[4096], c[4096], d[4096];
    int source1[8192], source2[8192];

    /* Loop A */
    /* Iterates over two 32KB */
    /* arrays of 8192 elements. */
    for(i=0; i<8192; i++){
        sum += source1[i] * source2[i];
    }

    /* Loop B */
    /* Iterates over five 16KB */
    /* arrays of 4096 elements. */
    for(i=0; i<4096; i++){
        result[i] = a[i] + b[i] + c[i] + d[i];
    }

    /* Loop C */
    /* Iterates over two 16KB */
    /* arrays of 4096 elements. */
    for(i=1; i<=10; i++){
        for(j=0; j<4096-(i*CACHE_LINE_SIZE); j++){
            result[j] += a[j+(i*CACHE_LINE_SIZE)];
        }
    }
}

```

Figure 3.1 : A Simple Function With Varying Data Cache Behavior

miss in the cache. With two cache ways enabled, all conflict misses between the two array references are eliminated. As the arrays are traversed, the cache lines accessed by each array reference are mutually disjoint, yielding a significant number of cache hits due to the spatial locality of the array accesses. Enabling more than two cache ways will not increase cache hit rate, as all conflict and capacity misses have been resolved in the two way enabled case, leaving only compulsory misses which cannot be eliminated.

The second loop, Loop B, performs the vector sum of four 16 KB arrays of integers. Again, the relative base addresses of the arrays are such that for every i , the i th element of each array will map to the same cache set. With only a single cache way enabled, all array references for a given loop index will map to the same cache line and thus evict each other before any reuse can be obtained. Even if all four cache ways are enabled, every memory reference in this loop will miss in the cache because the 5 array accesses conflict with each other. For any number of cache ways enabled, the number of total cache misses is thus the same. It should be noted that array padding between the *result* and *a* arrays can alleviate this behavior, in terms of reducing the miss rate with the cache fully enabled. Finding optimal array paddings for L1 data cache performance is interesting area of future research, but considered beyond the scope of this work.

The final loop, Loop C, exhibits both spatial and temporal locality among the array references. In this loop nest, two 16 KB arrays of integers are accessed in sequential order. The array references exhibit spatial locality across inner loop iterations and temporal locality across outer loop iterations. Again, the relative base addresses of the arrays are such that for every i , the i th element of each array will map to the same cache set. However, the array accesses are offset by at least a cache line for every loop iteration. Therefore, with a single cache way enabled, the spatial reuse across a cache line is captured for both arrays. The temporal reuse of the loop cannot be captured with a single cache way enabled, as references to the *result* array will evict data that was loaded for the *a* array in previous iterations of the inner loop. Similarly, references to the *a* array will evict data that was loaded for the *result* array in the previous iteration of the outer loop. With two cache ways enabled, both 16 KB arrays fit entirely into the cache, so that there will be no cache misses after the first outer loop iteration. Since a two-way set associative cache only suffers from compulsory misses in this loop, enabling additional cache ways provides no benefit.

Knowledge of the number of useful cache ways for a particular loop nest can be

exploited in a system with the ability to enable and disable cache ways independently. Simulating the execution of the code in Figure 3.1 on an appropriately configured version of SimpleScalar 3.0 [5] with the Wattch toolset [4] gives us hard data about its performance. Parameters used for simulation are detailed in Chapter 6. If the example above is run with the L1 data cache fully enabled during the entire programs execution, it has a first level data cache miss rate of 4.59%. If, instead, only two ways are enabled for the execution of Loop A, a single way is enabled for Loop B, and two ways are enabled for Loop C, the L1 data cache miss rate remains fixed at 4.59%. There is an decrease in the average number of instructions completed per clock cycle from 2.04 to 1.98 however, due to the flushing of data in L1 data cache out to L2 cache when cache ways are to be disabled. During the execution of the function, only one cache way is enabled for 10.22% of the time, two cache ways are enabled for 89.49% of the time, and during the remaining 0.28% of the time all four ways are enabled. Using SimpleScalar's Wattch tools to model power consumption, this results in a 62.82% reduction in average power consumed per clock cycle by the L1 data cache [5] [4]. In addition, average power per clock cycle for the entire processor is reduced by almost 10.90%. By detecting these cache sensitive regions in the program at compile time, we thus can reduce the amount of power consumed by the processor.

Chapter 4

Architecture

In order to reduce the power consumed by the data cache hierarchy, our proposed architecture incorporates a 4-way set associative L1 data cache that can be reconfigured dynamically at run time. Cache reconfiguration is achieved by independently enabling or disabling a given cache way, thus increasing or decreasing the overall L1 cache size. The decision to change the data cache size can be made either at run time by the hardware, or at compile time should enough information be available to the compiler. The hardware determines the appropriate data cache configuration by sampling L1 data cache and unified L2 cache miss rates. Based on the overall miss rate, the hardware attempts to minimize the amount of L1 data cache enabled while minimizing any increase in miss rate. Should enough information be available at compile time, the compiler can also insert hints in the program executable specifying the appropriate cache configuration for a given code region. The compiler focuses on loop nests, and makes an attempt to predict the minimal data cache size needed to reduce as many conflict and capacity misses as possible. By minimizing the number of misses in the L1 data cache, the compiler hopes to avoid costly access to the much larger and slower L2 cache. Additionally, if maximal L1 data cache hit rate can be achieved without enabling all of the L1 data cache ways, power can be saved by not accessing as many ways in parallel during cache lookups. Hint instructions containing information on the L1 data ways to enable during the loop nest's execution are then inserted in the binary which are detected by the hardware at runtime, and the data cache is reconfigured accordingly.

4.1 Reconfiguring the Cache via Hardware Performance Monitoring

In order to accurately reconfigure the L1 data cache at run time, the hardware must utilize knowledge of past and present cache performance to make predictions on future cache behavior. This is achieved by monitoring both L1 data cache and unified L2 cache performance throughout program execution. After every 50K instructions, miss rates for the L1 data cache and unified L2 cache are calculated for the 50K instruction stream. Decisions to reconfigure the L1 data cache are then made based on the cache performance for the last 150K instructions, or the last three sample points. This is similar to the methods used in by Balasubramonian in [2], and should require only a minimal number of transistors to perform the miss rate calculations and storing of the miss rate sample points. If the L1 data cache miss rate is below the miss threshold for the last three sample points and the L1 data cache is not already in the fully disabled state, i.e. direct mapped, then an L1 data cache way is disabled. The reasoning behind this is the following: if the majority of accesses to the L1 data cache are hits, then the working set of the program is most likely contained in L1 data cache and is perhaps much smaller than the amount of L1 data cache currently enabled. By disabling a way in the L1 data cache it may reduce the amount of power consumed on a given cache lookup, as well as the amount of leakage current lost on each clock cycle. In addition to this, it is desirable to contain the majority of the working set in L1 data cache so as to avoid costly accesses to the much larger and slower unified L2 cache. If the L1 data cache miss rate is above the miss threshold for any of the last three sample points, but the unified L2 miss rate is below the miss threshold for the last three sample points, and the L1 data cache is not fully enabled, then an L1 data cache way is enabled. The reasoning behind this is the following: either the working set of the program is exceeding the size of the L1 data cache, or there are a significant number of conflict misses occurring due to the associativity of the L1 data cache. Since the unified L2 miss rate is sufficiently low, the working set

must fit within the unified L2 cache. By enabling a way in the L1 data cache, it may contain more of the working set in the L1 data cache and thus reduce the number of costly accesses to the much larger unified L2 cache. If both the L1 data cache miss rate the the unified L2 cache miss rates are above the miss threshold for any of the past three sample points, then the L1 data cache is reset to fully enabled. The decision to reset the L1 data cache to fully enabled is made with the assumption that the working set of the program has changed and we are now incurring cold misses in the L1 and L2 caches. Since the miss rates already calculated pertain to a working set no longer being used, we pessimistically assume the new working set utilizes the full L1 data cache until we can prove otherwise.

4.2 Reconfiguring the Cache via Compile Time Information

For each compiler analyzable loop nest in the input program, the compiler must decide the appropriate amount of L1 data cache to enable during the loop nest's execution. Annotations are inserted into the program binary which inform the hardware as to how much L1 data cache to enable during execution of the given loop nest. As opposed to reconfiguring the cache via hardware performance monitoring, the compiler can set the L1 data cache associativity arbitrarily. That is to say, it does not need to incrementally enable and disable ways like the adaptive performance monitoring hardware. It is also important to note that the compiler must annotate both the start of the analyzable region and the end of the analyzable region. The annotation at the start of the analyzable region is necessary so that the L1 data cache can be configured appropriately during the execution of the code region. The annotation at the end of the analyzable region is necessary so that the hardware knows that the compiler no longer has information regarding cache performance of the code now executing. Under these conditions, the hardware reconfiguration scheme described in Section 4.1 can resume operation. The reasoning for locking out the hardware's ability to reconfigure the L1 data cache lies in the fact that the hardware has no knowledge of future

program behavior. The compiler sets the L1 data cache configuration in attempts to optimize the overall loop nest hit rate. This includes cold misses, as well as capacity and conflict misses. Many times at the start of a loop nests execution, a series of cold misses will occur in the L1 data cache until data is brought in from the lower cache levels. The compiler is aware of these cold misses, and realizes that increasing the L1 data cache size will do nothing to alleviate cold misses as they are independent of cache associativity. The hardware, on the other hand, will see these cold misses and think that either the working set exceeds the current size of the L1 data cache or that the working set has changed. In either case, it will naively try to increase the L1 data cache associativity in order to reduce the miss rate. By keeping the hardware from reconfiguring the L1 data cache during execution of compiler analyzed loop nests, this behavior can be avoided.

During program execution, if a cache configuration hint instruction is encountered the hardware prepares to enter a compiler analyzable region of code. The compiler hint instruction is decoded in order to determine the number of L1 data cache ways to enable. At this stage, the hardware performance monitoring mechanism is locked out from asserting changes in the L1 data cache configuration. Although unable to assert changes in the L1 data cache configuration, the hardware performance monitoring mechanism will still sample the cache miss rates as described in section 4.1. This is done so that once program execution exits the region analyzable by the compiler, an accurate view of the cache performance is available and the hardware performance monitoring mechanism can begin adapting to program behavior as soon as possible. After locking out the hardware performance monitoring mechanism, the hardware waits for all pending instructions in the pipeline to complete execution. This is necessary since any data residing in an L1 data cache way that is about to be disabled will be flushed out to L2 cache. We therefore let any instructions complete execution which are dependent on the data in L1 data cache ways about to be disabled. Once all instructions in the pipeline are finished executing, additional L1 data cache ways

are either enabled or disabled depending on the current configuration of the L1 data cache and the configuration that the compiler specified in the hint instruction. If the compiler hinted at having fewer cache ways enabled than were currently enabled, the data contained in any L1 data cache ways about to be disabled will need to be flushed out to the unified L2 cache. This is explained in greater detail in Section 4.3.

After setting the number of L1 data cache ways enabled to the value specified in the compiler hint instruction, execution continues with the given L1 data cache associativity. When the compiler hint specifying the end of the compiler analyzable region is found, execution is interrupted again. At this point, the L1 data cache is reset to full associativity, as it is pessimistically assumed that the majority of the data cache is used during most of the program runtime. The hardware reconfigurable scheme is again permitted to resize the L1 data cache based on the sampling of L1 data cache and unified L2 cache performance, and execution continues.

4.3 Enabling and Disabling Cache Ways

In order to maintain data coherency within the memory hierarchy, L1 data cache lines can not simply be enabled and disabled at will. As was mentioned in Section 4.1, care must be taken to ensure that the data contained in any valid cache lines that are about to be disabled are flushed to the unified L2 cache. In addition to this, any instructions in the pipeline before the flush occurs must be allowed to complete execution. This is due to the fact that in flight instructions may be dependent on data currently residing in an L1 data cache way which is about to be disabled and flushed to the unified L2 cache. If data were flushed from L1 data cache to the unified L2 before the instructions completed, then it would be possible for the data to then be reloaded into the L1 data cache when execution resumed. It should be noted that in some cases the data flushed from a given cache way is still valid. This is because the hardware shuts down the entire cache way, and does not necessarily keep the least recently used line of each set in the data cache. In such a case, when the next

memory reference occurs which refers to data contained in the recently flushed cache line occurs, the data will be reloaded into the L1 data cache and other less recently used data will be evicted. Such a settling down period can occur after a downsizing of the L1 data cache.

After flushing L1 data cache lines, the valid bits are marked as invalid so that no attempt to read data from the disabled cache line occurs. In order to cut off supply voltage and reduce leakage current in the disabled cache ways, a technique similar to the gated-Vdd method proposed in [19] [27] could be used. The idea behind this design is to introduce an extra transistor in the leakage path from the supply voltage to the ground of the cache's SRAM cells. For each cell in the cache ways that remain enabled, the extra transistor remains on. For those cache ways which are to be disabled, each cell's extra transistor is turned off. This, in effect, gates the cell's supply voltage and yields a much lower leakage current due to the two off transistors connected in series. By connecting the two off transistors in series, a self reverse-biasing effect is created which is referred to as the stacking effect [24]. See Figure 4.1 for details. Once the L1 data cache lines to be disabled have the gated-Vdd transistor turned off and the valid bits marked as invalid, execution can resume.

The process of enabling a given L1 data cache way is somewhat simpler. We must assert that the valid bits for the cache ways to be enabled remain invalid upon enabling. Additionally, the gated-Vdd transistors for each cell in the cache ways to be enabled need to be turned on. There is no need to purge instructions from the pipeline as is the case in disabling a cache way, as no valid data remains in the cache ways which we are enabling. Once power returns to the previously disabled cache ways, program execution can resume.

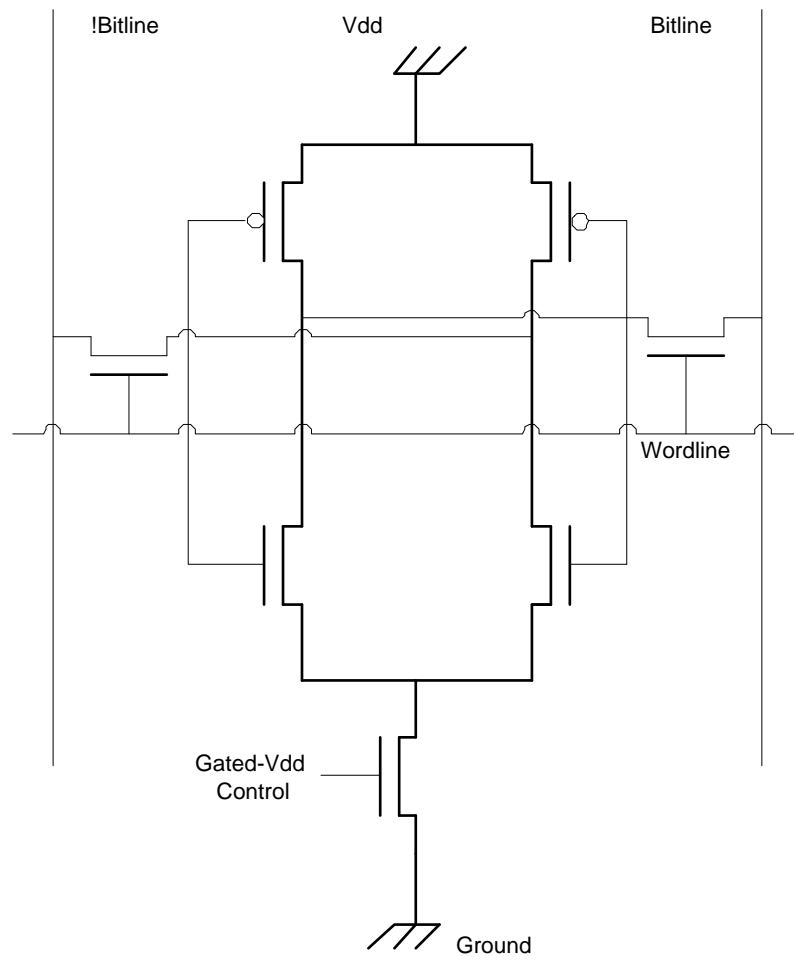


Figure 4.1 : SRAM with an NMOS Gated-Vdd

Chapter 5

Compiler

In order to accurately determine the number of L1 data cache hits and misses that occur over a given region of code, the compiler must obtain vast amounts of information about the program's behavior. Our efforts in compiler analysis were focused on loop nests within the program, due to their somewhat regular structure and predictable control flow. The algorithms described in this chapter are an extension of the work done by Somnath Ghosh in his Ph.D dissertation at Princeton in 1999 [9] [11]. The subsections that follow describe the process of determining the proper L1 data cache requirements of analyzable loop nests. Sections 5.1, 5.2, 5.3, and 5.4 all describe the preliminary work done by Ghosh in his dissertation, while Section 5.5 describes the extensions we made to his work in order to predict L1 data cache requirements of loop nests.

5.1 Analyzable Loop Nests

In order to accurately determine the number of L1 data cache hits and misses that occur during execution of a loop nest, our algorithm must be able to build an exact model of the programs behavior within the loop nest. Since an exact model is necessary to predict L1 data cache performance, not all loop nests will be analyzable, as exact information about the program is not always available at compile time. The first requirement of the compiler is that control flow be predictable within the loop nest. Since control flow must be predictable, only loop nests that are perfectly nested, or loop nests that contain a single basic block of control flow between successive nests can be considered analyzable. If control flow were not predictable, then at any given

point in the iteration space of the loop nest, the exact sequence of memory references which occurred would be unknown. Without the exact stream of memory references that occurred up to the current point in the iteration space, exact contents of the L1 data cache becomes indeterminate. From such an indeterminate state, it would then be impossible to determine whether subsequent memory references were hits or misses in the L1 data cache. It should be noted that an approximation of a loop nest's L1 data cache miss rate could be obtained in cases which control flow is not entirely predictable. Using hot paths through loop nests would most likely provide an approximation of the L1 data cache performance. We consider this to be an interesting avenue of future work, but it is beyond the current scope of our research.

In addition to predictable control flow, each loop within the analyzable nest must have loop bounds that can be determined at compile time. Loop bounds can either be constant expressions or affine combinations of the enclosing loop indices. If loop bounds are not known at compile time, the exact iteration space of the loop nest becomes unknown. With an unknown iteration space, it then becomes impossible to determine capacity misses and certain types of conflict misses which occur within the loop nest. It is important to note that the loop bounds are not always necessary in determining the exact L1 data cache miss rate of loop nests. An example of this is *Loop A* and *Loop B* in the example code shown in Figure 3.1, in which there is only spatial reuse along a cache line. This, again, is an interesting point, but considered beyond the scope of our research.

Loop index variables and array subscript expressions must also be affine combinations of the enclosing loop indices, which is a common model for research in compiler memory analysis. Scalar variables are considered a special case of 1-D arrays.

The final restriction on compiler analyzable loop nests is that the relative base addresses of arrays accessed within the loop nest be known at compile time. Such knowledge is necessary for the compiler to determine when conflict misses occur between two distinct array references in the loop nest. Simply evaluating the loop index

variable for a given point in the iteration space is not enough to determine such conflict misses, as changes in the relative base addresses of two arrays will alter which cache line a given array element maps to. It is important to note, however, that the absolute base address of any array is never needed. Simply knowing the relative base addresses of arrays with respect to each other is enough to determine whether or not a conflict miss occurs between two successive array accesses. This concept is explained in further detail in Section 5.2. Although beyond the scope of this body of work, solving for optimal array base addresses at compile time is certainly a reasonable idea [9] [23]. At compile time, the relative array base addresses which result in the minimal number of conflict misses could be calculated. Positioning hints could then be passed on to the linker such that array data is padded to create the appropriate relative base addresses.

5.2 Cache Miss Equations: What They Represent

The set of cache miss equations, or CMEs, generated by the compiler models a loop nest’s reference stream and cache conflict patterns in a mathematical framework that can be analyzed at compile time. The algorithm described here is an extension of the original CME framework developed by [9] which used solutions to CMEs to determine appropriate array padding and tile sizes during compiler loop transformations. Rather than use CME solutions to modify the code shape of loop nests in an attempt to optimize L1 data cache performance, the algorithm described in this section approaches the problem from a different angle. Solutions to CMEs are used in determining the minimal amount of L1 data cache associativity necessary to capture the majority of capacity and conflict misses that occur in a given loop nest. By enabling only the minimal amount of associativity necessary to capture the capacity and conflict misses within the loop nest, additional power is not consumed on L1 data cache lookups by accessing additional cache ways in parallel. Additionally, since the majority of capacity and conflict misses are realized in the L1 data cache, additional power is not

wasted on costly accesses to unified L2 cache.

5.2.1 Terminology

The work with CMEs presented in this section is derived from the body of research in which iteration spaces and reuse vectors are used to model memory hierarchy behavior for dependence analysis [11] [21]. The following section describes the terminology used in generating the CMEs in Section 5.3 and solving the CMEs in Section 5.4.

The range of induction variable values traversed by a loop nest is called an *iteration space*. Each distinct iteration of the loop nest is a single entity called an *iteration point*, whereby the set of all iteration points constitutes the iteration space. Every iteration point is identified by its *index vector* $\vec{i} = (i_1, i_2, \dots, i_n)$, where i_l is the loop index of the l^{th} loop in the nest with the outermost loop being represented by the first dimension. It is also important to be able to describe the ordering in which various iteration points occur. In a situation where iteration point \vec{p}_2 executes after iteration point \vec{p}_1 , we write $\vec{p}_2 \succ \vec{p}_1$ and say that \vec{p}_2 is lexicographically greater than \vec{p}_1 .

We refer to a static read or write in the program source code as a *reference*. The actual execution of such a read or write at runtime is referred to as a *memory access*. A *memory line* refers to a cache line sized block in memory, and a *cache line* refers to the physical line in data cache that a memory line maps to. In the deriving of CMEs, we refer to the total cache size as C_s , and the associativity of the cache as k . The size of a cache line in bytes is represented by L_s , while the number of cache sets is represented by N_s . Since each cache set is made up of k cache lines, the total cache size can be represented by Equation 5.1. The cache set accessed by a given reference R_A at iteration point \vec{i} is given by Equation 5.2, with everything in units of data element size. $Mem_{R_A}(\vec{i})$ is an affine function of the loop indices and can be computed from the subscript expressions of R_A .

$$C_s = N_s \times k \times L_s \quad (5.1)$$

```

DO i = 1, N
  DO k = 1, N
    DO j = 1, N
      Z(j,i) += X(k,i) * Y(j,k)
    
```

Figure 5.1 : Simple Matrix Multiply Loop Nest

$$Cache_Set_{R_A}(\vec{i}) = \lfloor Mem_{R_A}(\vec{i})/L_s \rfloor \bmod N_s \quad (5.2)$$

As an example, the cache set of the reference $Z(j, i)$ in the matrix multiply loop nest of Figure 5.1 is given by Equation 5.3. All numbers are in units of data element size, the base address of the array Z is 4192, and the number of elements per column of the Z array is 32. The data cache used is an 8KB 2-way associative cache with 128 cache sets and 4 data elements per cache line.

$$Cache_Set_Z(j, i) = \lfloor (4192 + 32i + j - 1)/4 \rfloor \bmod 128 \quad (5.3)$$

Reuse vectors are used to represent repeated memory access patterns in loop based codes [26]. If a given memory reference accesses the same memory line in iterations \vec{i}_1 and \vec{i}_2 , then there exists a reuse in direction $\vec{r} = \vec{i}_2 - \vec{i}_1$. We therefore refer to \vec{r} as the reuse vector. When a given reference reuses a memory line which was previously accessed, we need to know the iteration point at which the memory line was last accessed, and the reference that accessed it. Using this information, we can determine whether another reference accessed a memory line which mapped to the same cache line and thus evicted the data from L1 data cache resulting in a cache miss. If the reuse results in a cache hit, we say that the reuse has *locality*. This is the main idea behind the CMEs. We find loop iterations in which a given reuse does not result in a data cache hit.

5.3 Generating Cache Miss Equations

The compiler algorithm in this section divides the CMEs into two distinct types: *cold miss equations* or *cold CMEs* and *replacement miss equations* or *replacement CMEs*. The solutions to the *cold CMEs* represent potential *cold* or *compulsory misses*, that is to say misses that occur on the first reference to a memory line. Solutions to the *replacement CMEs* represent all other misses, including both *capacity* and *conflict* misses.

During the forming of cold CMEs, each loop nest is treated in isolation. The L1 data cache is assumed to be empty at the start of loop nest execution, and any intra-loop data dependencies are ignored and considered beyond the scope of this work. The simplest type of cold CME is of the form $(i \bmod 4) = 0$, representing a sequence of unit stride accesses in which four data elements fit in a given cache line. After every four accesses, a cold miss will result as the data accessed now maps to a new cache line. Forming the cold CMEs becomes more complex depending on nesting depth, strides, and array offsets, but the underlying concept remains the same. As a result, cold memory accesses occur at the iteration points that contain either the first access along the direction of the reuse vector, or accesses that just crossed a memory line boundary along the direction of the reuse vector.

Replacement CMEs represent capacity and conflict misses, which occur when the currently accessed memory line was previously in the cache but has since been evicted by a reference to another memory line which maps to the same cache line. We'll first consider CMEs in a direct mapped cache, and then extend the idea to caches of arbitrary associativity. A miss occurs in the direct mapped case if between successive accesses to a given memory line $Line_1$, another access occurs to a distinct memory line $Line_2$ which maps to the same cache line as $Line_1$. Consider the reference stream R_A, R_B, R_A , in which a conflict occurs if the cache line accessed by R_A is the same as the cache line accessed by R_B . This occurs if the address referenced by R_B is a non-integer multiple of the cache size and within the cache line size range. The cache

line size range is included to capture the situations in which the memory addresses do not differ by an exact multiple of the cache size, but map to the same cache line. In the case of the k -way associative cache, a conflict occurs if between consecutive accesses to a particular memory line at least k other access occur to distinct memory lines which map to the same cache set.

5.3.1 Example: Forming CMEs

Equations 5.4, 5.5, and 5.6 illustrate the process of forming CMEs for $Z(j, i)$ in the example shown in Figure 5.1. The array column size $N = 32$, and the iteration point is $\vec{i} = (i, k, j)$. For an 8KB 2-way set associative cache with 128 cache sets and 4 array elements per cache line, the equations below show the replacement CMEs for the interferences with $X(k, i)$. 4192 and 2136 are the base addresses in array elements of arrays Z and X , and $n > 0$.

$$Cache_Set_Z(j, i) = Cache_Set_X(k, i) \quad (5.4)$$

$$\lfloor (4192 + 32i + j - 1)/4 \rfloor \bmod 128 = \lfloor (2136 + 32i + k - 1)/4 \rfloor \bmod 128 \quad (5.5)$$

$$4192 + 32i + j = 2136 + 32i + k + 512n \quad (5.6)$$

5.4 Solving the Cache Miss Equations

Once the cold and replacment CMEs are generated, it is necessary to solve them in order to find the number of cold, capacity, and conflict misses that occur during execution of the loop nest. For each reference in the loop nest, we generate a set of equations for each of the reference's reuse vectors. For each reuse vector, there can exist at most two cold CMEs representing cold misses along that vector [10], as well as replacement CMEs representing the self and cross interferences of this reference

with itself and others. It is important to note that CME solution points represent only *potential* cache misses; to find the actual cache misses one must consider the effects of multiple reuse vectors at once.

The algorithm first sorts the reuse vectors of a reference in lexicographically increasing order. For each of the sorted reuse vectors, a number of CMEs are generated, each of which produces a collection of CME solution points. The algorithm investigates one reuse vector at a time, starting from the shortest one first. After investigating a reuse vector, some CME solution points are declared definite miss points, while others are marked indeterminate. If any iteration point \vec{i} is a solution for a cold CME of the current reuse vector \vec{r} , then there is a cold miss along reuse vector \vec{r} at iteration point \vec{i} . Thus there exists no reuse along \vec{r} at that point. As we cannot take any further decision about these iteration points without considering other reuse vectors, we declare them as indeterminate for the current reuse vector \vec{r} . These indeterminate points are then passed on to the next reuse vector for further investigation. If an iteration point is not a solution for a cold CME, but is a replacement miss along the current reuse vector \vec{r} , it is declared a definite miss point for the reference. Finally, any iteration point which is neither a cold CME solution point nor a replacement miss along the current reuse vector \vec{r} is a guaranteed cache hit. The algorithm continues investigating further reuse vectors until the number of indeterminate points either goes to zero, or is sufficiently small as defined by a user threshold. At that point we can stop the process.

5.5 Cache Miss Equations to Cache Configuration

As was introduced at the start of Chapter 5, the compiler algorithm described attempts to determine the minimal amount of L1 data cache associativity necessary to realize the minimum number of capacity and conflict misses that occur in a given loop nest. Only a minimal number of cache ways are enabled, so as to minimize the amount of power consumed on a given L1 data cache lookup by not accessing

additional cache ways in parallel. Additionally, the compiler must be sure to provide enough L1 data cache associativity to prevent the majority of conflict and capacity misses from occurring in the loop nest. By preventing such L1 data cache misses, costly accesses to the much larger, and slower L2 unified cache can be avoided. The algorithm to determine proper L1 data cache associativity for a given loop nest works as follows: The compiler finds loop nests in the program which meet the criteria previously described in Section 5.1. Once the analyzable nests are found, the compiler determines the order in which array and scalar memory references in the loop nest occur. The ordering of memory references is necessary in determining conflict misses between distinct memory references. It should also be noted that compiler naively assumes that the memory references inside the loop nest will be executed in the same order in which they occur in the code. Subsequent compiler passes such as instruction scheduling may rearrange the order in which some memory references in the loop nest occur. Additionally, modern superscalar processors with out-of-order execution may dynamically execute the memory reference instructions a different order than was originally analyzed by the compiler. Preliminary studies performed on SimpleScalar's out-of-order simulator, however, showed such side effects to be negligible.

Once the ordering of memory references inside the loop nest is found, the reuse vectors, cold CMEs, and replacement CMEs are formed for each reference. The L1 data cache on our target machine is 4-way set associative with a total size of 64KB when all four cache ways are enabled. By disabling the various cache ways, it can be reconfigured to any one of the four following states: 16KB direct mapped, 32KB 2-way associative, 48KB 3-way associative and 64KB 4-way associative. The algorithm as described in Section 5.4 is used to determine the number of cold, capacity, and conflict misses that occur in the each loop nest. The cache misses are calculated for each reference in a given loop nest, and for each loop nest the CMEs are solved for each possible L1 data cache configuration our target machine supports. The algorithm to solve the CMEs as described in Section 5.4 only supports direct mapped caches,

so in order to approximate the behavior of the target machines set associative cache, the each loop nest's CMEs are solved using a direct mapped cache equivalent to the total size of a given set associative configuration. As an example, to determine the cache misses of a given loop nest with three L1 data cache ways enabled on our target machine, the loop nests CMEs are solved using a 48KB direct mapped cache as to approximate the behavior of our 48KB 3-way enabled configuration.

Once the loop nest's cold, capacity, and compulsory misses are calculated for each possible L1 data cache configuration our target machine supports, the compiler must determine the proper number of L1 data cache ways to enable. Nothing can be done to decrease the number of cold misses which occur in a given loop nest, as the compiler can only control the set associativity of the L1 data cache and cold misses are independent of associativity. Instead, the compiler attempts to capture the majority of capacity and conflict misses that occur in the loop nest, while enabling only the minimal number of L1 data cache ways to do so. By doing this, additional power will not be consumed on L1 data cache accesses by accessing multiple ways in parallel, and many costly accesses to the larger and slower unified L2 cache will be avoided. The compiler thus totals the number of capacity and conflict misses that occur in a given loop nest for each set associativity supported by the target machine. It then chooses the set associativity for which the minimum number of conflict and capacity misses are realized. In many cases, however, providing additional associativity does not reduce the number of L1 data cache misses that occur in a given loop nest. In such cases, the minimal set associativity is chosen such that the total number of capacity and conflict misses that occur does not increase. Once the appropriate L1 data cache set associativity is determined, special instructions are inserted at the head and tail of the loop nest, as was described in Section 4.2. The instruction at the head of the loop nest tells the hardware how many L1 data cache ways to enable during execution of the loop nest, and subsequently locks out the hardware reconfiguring mechanism. The instruction inserted at the tail of the loop nest tells the

hardware that the execution has entered a region of code for which the compiler has no knowledge of cache performance. As a result, the adaptive hardware reconfiguring mechanism is once again enabled.

Chapter 6

Experimental Setup

All results were obtained using the SimpleScalar 3.0 toolset, available for download at www.simplescalar.com [5]. This provided a framework in which runtime performance could be easily monitored. In order to obtain data on processor power consumption, the Wattch toolset was used with SimpleScalar as well, available for download at www.ee.princeton.edu/~dbrooks/ [4]. Simulations were run using either the included out-of-order execution simulator or the cache simulator. The simulators were modified in order to support a reconfigurable L1 data cache as was described in Chapter 4. Additionally, instructions were added to represent L1 data cache configuration hints inserted by the compiler. Source code for the supplied simplescalar target compiler (gcc 2.6.3) was modified to support the cache hint instructions as well. All programs were compiled and simulated using the PISA instruction set, as is described in [5].

6.1 Simulated Architecture

Table 6.1 lists the parameters used in our architectural simulations. The value of 5 clock cycles for the L1 data cache way enable and disable time is used as a safely pessimistic value based on the figures in Falsafi et. al [19] [27].

6.2 Tool Chain

As was mentioned in Chapter 5, the compiler pass to analyze loop nest cache usage was built as an extension to the previous CME work done by Ghosh et. al [9]. Compiler passes to generate the CMEs were written using Suif 1.3.0.5 [25]. Figure 6.1 depicts the tool chain used to create a SimpleScalar executable with loop nest cache hints

L1 Instruction Cache	Fixed size, 128KB direct-mapped, 32 byte line size, LRU replacement policy, 1 cycle latency
L1 Data Cache	Runtime reconfigurable, 2KB direct-mapped, 4KB 2-way assoc, 6KB 3-way assoc, 8KB 4-way assoc, 32 byte line size, LRU replacement policy, 1 cycle latency
L2 Unified Cache	Fixed size, 256KB 4-way assoc, 64 byte line size, LRU replacement policy, 6 cycle latency
CLOCK SPEED	600 MHz
Clocking Method	Aggressive conditional, with leakage current
Integer ALUs	4 + 1 integer multiply
Floating Point ALUs	4 + 1 floating point multiply
Memory ALUs	2
Decode / Issue / Commit Width	4 instructions
Register Update Unit	16 entries
Load / Store Queue	8 entries
Branch Predictor	bimodal, 2KB entries + 8 entry return buffer, 512 entry BTB
TLB	4KB, 4-way assoc, LRU replacement policy
Cache Way Enable and Disable Time	5 cycles or 8.33 nSec

Table 6.1 : Simplescalar Simulation Parameters

from a given input program. The starting C or Fortran source code is converted to Suif intermediate format by the Suif front end. From this stage, the compiler finds loops in the control flow graph which meet the criteria for analyzability as described in Section 5.1. At this stage, a *loop nest ID* file is created which records where in the original source code analyzable loops are located. Once the loop locations are recorded, our compiler pass proceeds to create the CMEs for each analyzable loop nest in the code. Once the CMEs are generated, they are written to an intermediate file, as is depicted in the *CME* stage of Figure 6.1. It is important to note that the actual CMEs are not solved by Suif itself. Suif only forms the equations based on compile time information obtained for each analyzable loop nest in the code.

Once the CMEs are generated, the *CME Solver* determines the number of cold, capacity, and conflict misses for each loop nest as is described in Section 5.4. For each independent loop nest, the miss statistics are generated for each possible L1 data cache configuration supported by our target machine and written to file as is depicted in the *Loop Nest Cache Miss Statistics* stage. Using the miss statistics for each loop nest, a postprocessing tool decides the appropriate amount of L1 data cache to enable during the loop nest's execution. Once the proper L1 data cache associativity is determined for each loop nest, the data is written to file. At this stage, the *Loop Nest ID* file, *L1 Data Cache Config* file, and original input source code are input to the *Add Cache Config Hint* stage, which inserts cache configuration hints into the original source code for each analyzable loop nest. Once the source code is annotated with the cache configuration hints, it is compiled by our modified version of SimpleScalar's PISA compiler and the resulting cache hint annotated SimpleScalar executable is created.

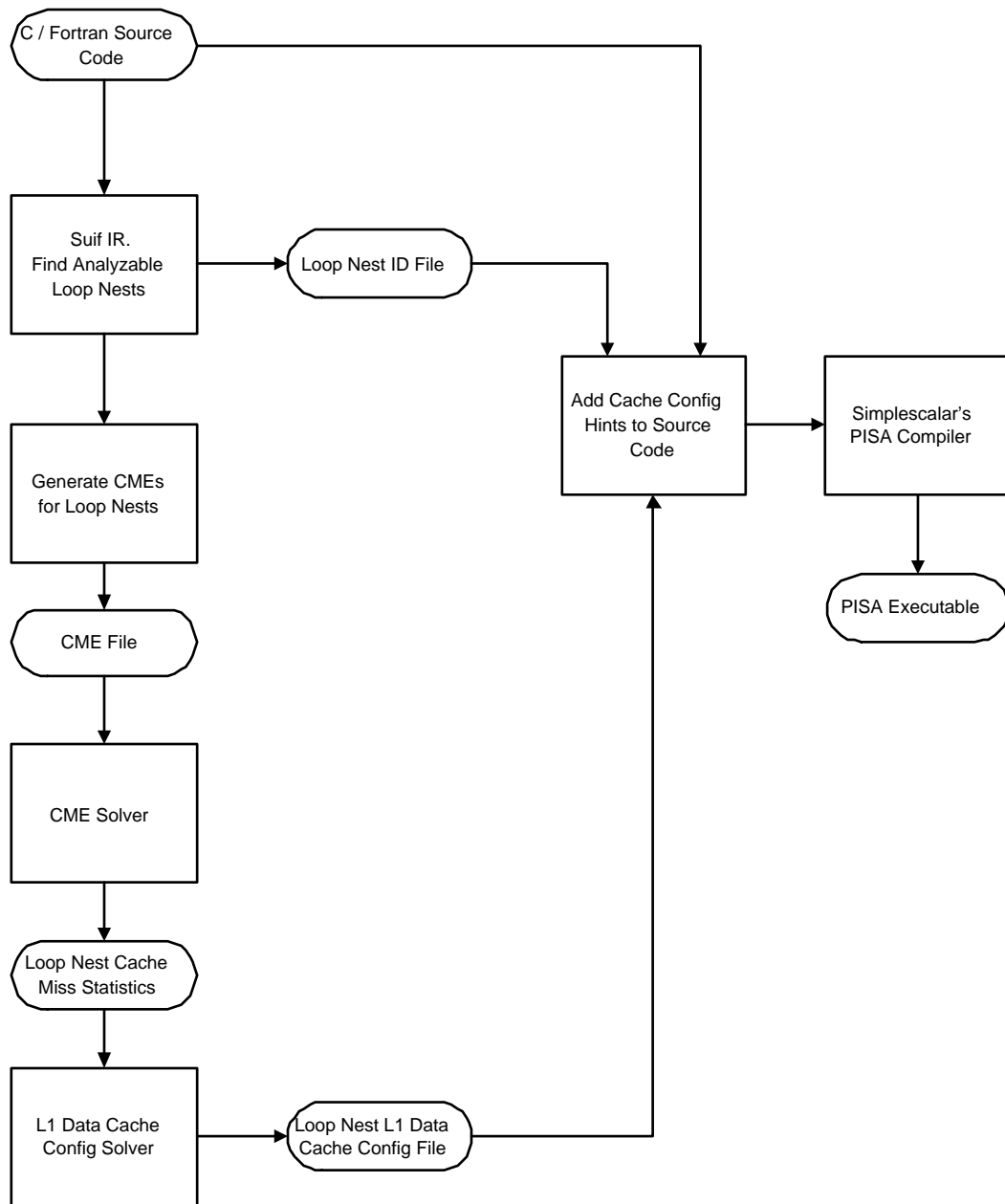


Figure 6.1 : Block Diagram of Tool Chain

Chapter 7

Results

In this chapter, we present the results obtained by reconfiguring the L1 data cache via hardware performance monitoring and reconfiguring via compile time analysis, as was presented in Chapters 4 and 5. Section 7.1 discusses how the data was obtained, while Section 7.2 illustrates the baseline cache performance of each benchmark simulated, when the L1 data cache size is fixed throughout program execution. Section 7.3 shows the relative cache performance when the L1 data cache size is set dynamically at runtime using the hardware performance monitoring mechanism only. Finally, Section 7.4 shows the effectiveness of our compile time analysis of loop nest cache performance, as was described in Chapter 5.

7.1 Simulation Details

The SPEC CPU2000 and Mediabench benchmarks were used to evaluate the effectiveness of reconfigurable data caches for reducing power consumption [1] [15]. All simulations were performed using a modified version of SimpleScalar’s cache simulator, as was described in Chapter 4. The reduced input data sets created by Kleinosowski were used for each of the SPEC CPU2000 benchmarks [14]. For many of the programs in the SPEC CPU2000 benchmark suite, the L1 and L2 data cache performance differs when using the reduced input data sets as opposed to the standard SPEC CPU2000 input data sets. Specifically, the relative L1 and L2 miss rates are significantly lower as the total working set size is smaller with the reduced input set. Refer to the SPEC CPU2000 documents for further details [1] [6] [14] [17]. As a result, the simulations presented in this section have been run with a significantly smaller L1 data cache

size than is typically found in a modern microprocessor. The reconfigurable L1 data cache simulated is an 8KB, 4-way set associative cache with 32 byte line size. It can be reconfigured to 6KB 3-way set associative, 4KB 2-way set associative, and 2KB direct mapped data, as is described in Chapter 4.

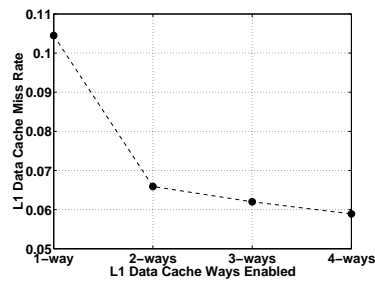
7.2 Baseline Cache Performance

In this section, the baseline performance of each benchmark is presented, where the L1 data cache size is statically set at program start time. Figure 7.1 and Figure 7.3 show the L1 data cache and unified L2 cache performance of each SPEC CPU2000 benchmark respectively. The same data is shown for the Mediabench benchmarks in Figure 7.2 and Figure 7.4. In each of the Figures 7.1 – 7.4, the L1 data cache size is statically set at program start time as is shown along the x-axis. The relative L1/L2 miss rate is shown along the y-axis. This was done so as to simulate the data cache performance using a standard non-reconfigurable L1 data cache. Note that the last data point along the x-axis, 8KB 4-way associative, is the equivalent of leaving our dynamically reconfigurable L1 data cache fully enabled during program execution.

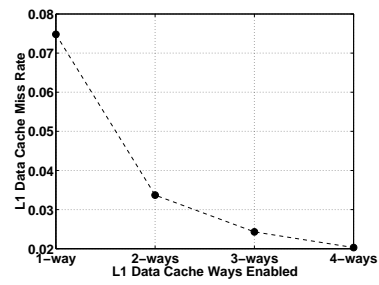
For each of the benchmarks simulated, the L1 data cache miss rate increases as the amount of L1 data cache available decreases, as is to be expected. Some of the benchmarks such as 177.mesa, 179.art and Adpcm do not show much increase in L1 data cache miss rate until the L1 data cache size is decreased to about 2KB. This is due to the fact that the working set of these benchmarks is relatively small, and tends to fit into an L1 data cache of size greater than or equal to approximately 4KB. As the L1 data cache size approaches 2KB however, all of the benchmarks simulated exhibit a noticeable increase in the L1 data cache miss rate. This is due to the fact that with only 2KB of available L1 data cache, the majority of the working set does not fit into L1 data cache at any given time. This results in an increased number of capacity and conflict misses which drive the miss rate up, resulting in accesses to the unified L2 cache. This is the behavior that we hope to avoid in using the reconfigurable L1

data cache. It is true that with only 2KB of L1 data cache enabled, significant power savings will be achieved in the L1 data cache over the fully enabled case. In almost all cases, however, the power savings achieved in the L1 data cache will be offset by increased power consumption in the unified L2 cache. This is because as the L1 data cache size decreases below the size of the program's working set, increasing L1 data cache accesses will result in capacity and conflict misses. Such misses will then result in an access to the much larger unified L2 cache. Although the power consumed on the initial access to the reconfigurable L1 data cache will be less than that of an access to a fully enabled L1 data cache, the subsequent access to the unified L2 cache will consume additional power.

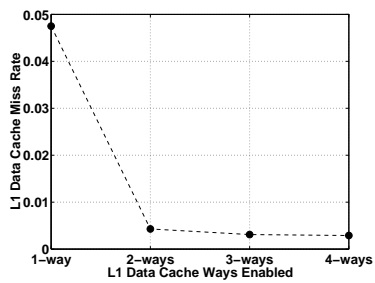
Looking at Figures 7.3 and 7.4, it may seem counterintuitive that as the L1 data cache miss rate increases, the corresponding unified L2 cache miss rate decreases. Let us first consider the case when the working set of the program fits in L1 data cache, and the miss rate of the unified L2 cache is relatively high. In this case, the majority of data cache accesses falling through to the unified L2 are cold misses. Since the working set of the program is fitting into L1 data cache, the majority of potential capacity and conflict misses are being realized in L1 data cache. Although the miss rate for the unified L2 cache is relatively high, the total number of accesses falling through to the unified L2 is relatively low. As the size of the L1 data cache is decreased, the working set of the program no longer fits in L1 data cache. The number of capacity and conflict misses that occur in L1 data cache thus increases. This increase in capacity and conflict misses as the L1 data cache level results in a marked increase in accesses to the unified L2 level. Since the unified L2 cache is much larger than the L1 data cache, quite often the working set of the program will fit into unified L2 cache. The decrease in unified L2 miss rate is a result of the increasing number of accesses which now miss in the L1 data cache, and subsequently hit in the unified L2 cache. Although the unified L2 miss rate is lower, the total number of accesses to the unified L2 is now much higher than with a larger L1 data cache.



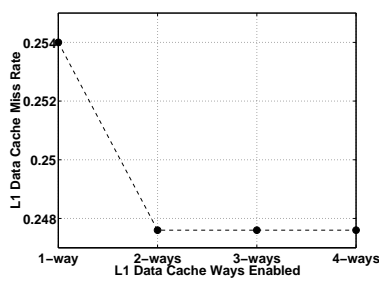
(a) 164.zip



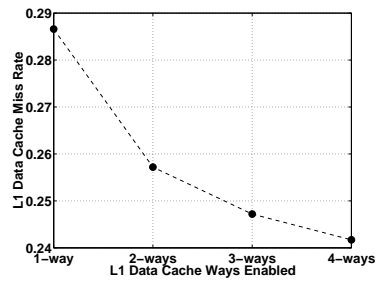
(b) 175.vpr



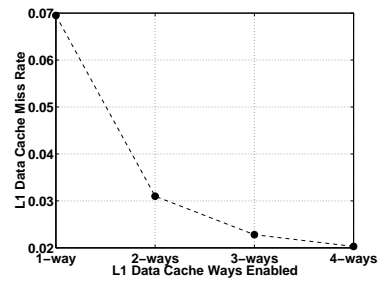
(c) 177.mesa



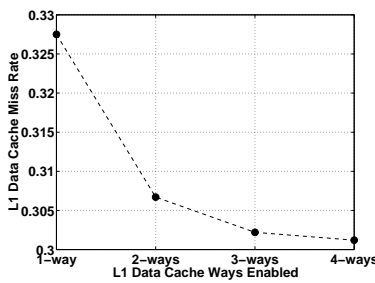
(d) 179.art



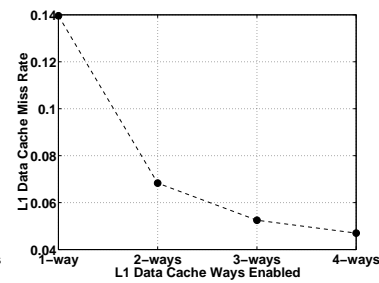
(e) 181.mcf



(f) 183.equake

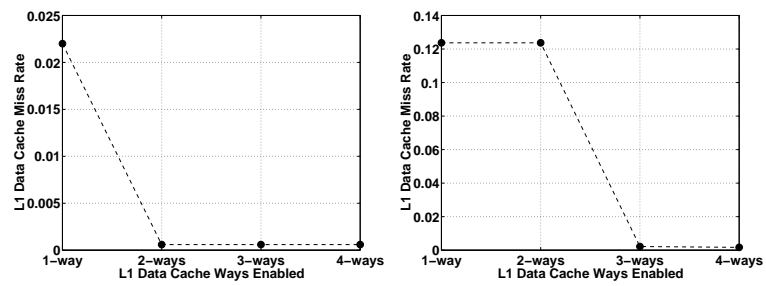


(g) 188.ammp



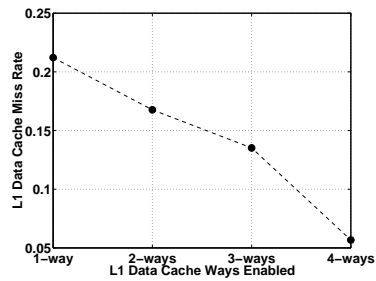
(h) 197.parser

Figure 7.1 : SPEC CPU2000 Fixed L1 Data Cache Size, L1 data cache miss rates



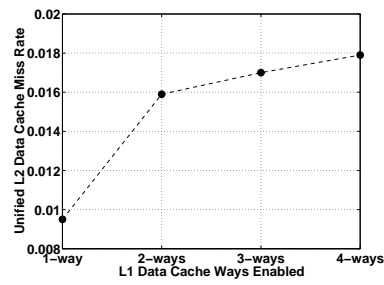
(a) Adpcm

(b) Mpeg2 (encode)

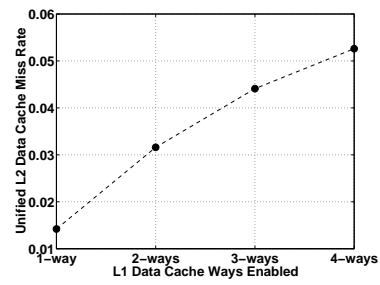


(c) Epic

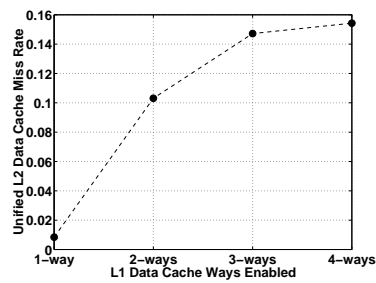
Figure 7.2 : Mediabench Fixed L1 Data Cache Size, L1 data cache miss rates



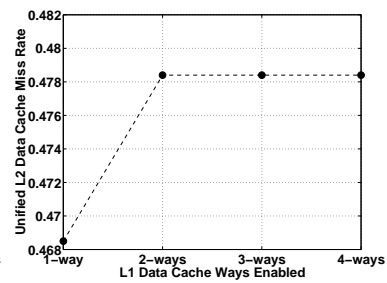
(a) 164.zip



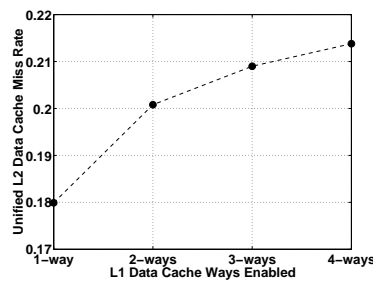
(b) 175.vpr



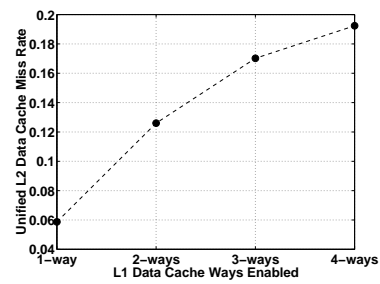
(c) 177.mesa



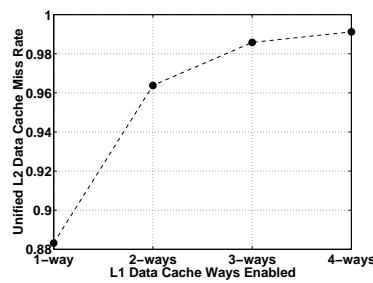
(d) 179.art



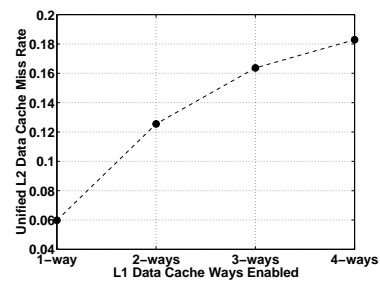
(e) 181.mcf



(f) 183.equake

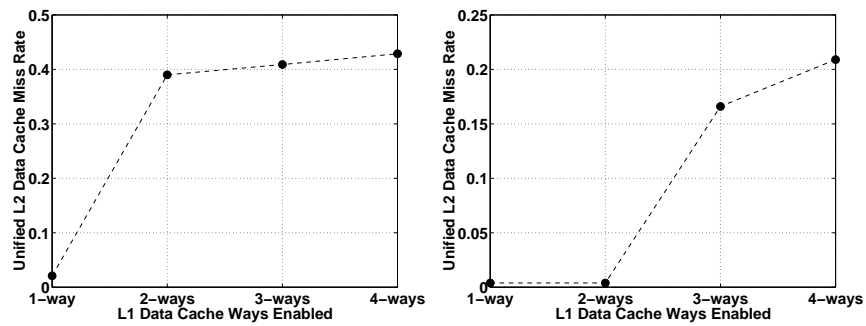


(g) 188.amm



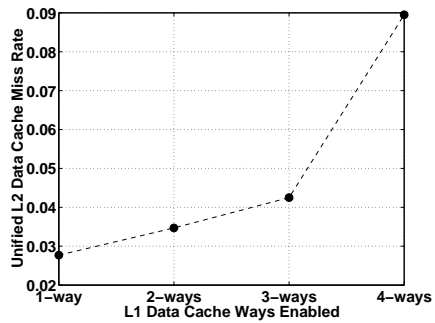
(h) 197.parser

Figure 7.3 : SPEC CPU2000 Fixed L1 Data Cache Size, Unified L2 cache miss rates



(a) Adpcm

(b) Mpeg2 (encode)



(c) Epic

Figure 7.4 : Mediabench Fixed L1 Data Cache Size, Unified L2 cache miss rates

In Figures 7.1 and 7.2, there is a distinct knee in each graph where the L1 data cache miss rate increases drastically. For 164.zip, 177.mesa, 179.art, and Adpcm it clearly lies at a 4KB two-way associative L1 data cache. For the other benchmarks, it is somewhere between a 4KB two-way and 6KB three-way associative L1 data cache. If power is to be saved by reconfiguring the L1 data cache size, then the majority of runtime should be spent with the L1 data cache sized to the forementioned knee in Figures 7.1 and 7.2. By doing this, only the minimal amount of L1 data cache will be enabled to maintain cache hit rate. Power can be saved by not accessing as many L1 data cache ways in parallel during a cache lookup, and leakage current in the disabled ways will be drastically reduced as was described in Section 4.3. Additionally, by maintaining hits in the L1 data cache, costly accesses to the much larger and slower unified L2 cache can be avoided. In the next section, the L1 data cache is resized dynamically at runtime using the hardware performance monitoring mechanism described in Section 4.1.

7.3 Reconfiguring the Cache Via Hardware Performance Monitoring

This section contains results for when the L1 data cache is dynamically reconfigured at runtime using the hardware performance monitoring mechanism described in Section 4.1. As was discussed in Section 7.2, there clearly exists a given L1 data cache size below which L1 data cache miss rate increases. It was at this L1 data cache size, that the hardware performance monitoring mechanism would ideally operate during the the majority of program execution. By doing this, power would be saved by having a number of L1 cache ways disabled during program runtime. Additionally, since there was little or no increase in the L1 data cache miss rate, the number of costly accesses to the larger and slower unified L2 cache would be minimal. Table 7.3 shows the L1 data cache and unified L2 cache miss rates, as well as the percentage of runtime spent in each L1 data cache state when the L1 data cache is reconfigured

Benchmark	L1 miss rate	L2 miss rate	% 1-way enabled	% 2-ways enabled	% 3-ways enabled	% 4-ways enabled
164.zip	0.0620	0.0169	1.70	4.17	7.15	86.98
175.vpr	0.0259	0.0397	4.64	15.62	21.85	57.88
177.mesa	0.0106	0.0388	16.53	16.53	16.53	50.41
179.art	0.2476	0.4784	0.04	0.05	0.05	99.86
181.mcf	0.2419	0.2137	0.73	0.37	0.37	99.54
183.earthquake	0.0280	0.1397	13.19	14.22	14.14	58.46
188.ammp	0.3034	0.9807	3.13	3.27	3.26	90.36
197.parser	0.0477	0.1800	0.22	1.38	1.79	96.62
adpcm	0.0188	0.0243	84.69	3.06	3.06	9.18
mpeg2	0.0233	0.0213	19.95	21.06	20.74	38.24
epic	0.0626	0.0816	19.34	5.20	7.20	68.26

Table 7.1 : Reconfigurable L1 Data Cache via Hardware Sampling, L1 and L2 Cache Performance

at runtime using hardware performance monitoring only. No compile time information is used in these simulations. Table 7.3 shows the number of instructions in the simulation, and the number of times an L1 data cache way is enabled, disabled, or the L1 data cache size is reset. Finally, Table 7.3 shows the relative increase in L1 data cache miss rate when the L1 data cache size is reconfigured at runtime versus leaving the L1 data cache fully enabled during the programs execution. Looking at data in Table 7.3 and Table 7.3, it is clear that the hardware performance monitoring mechanism does not always operate at the optimal configuration as was previously described.

Looking at 164.zip, 179.art, 181.mcf, 188.ammp, and 197.parser in Table 7.3, it can be seen that the L1 data cache remains fully enabled for the majority of program runtime. For each of these benchmarks, at least 85% of the total program runtime was spent with all of the reconfigurable L1 data ways enabled. Referring back to Figures 7.3 and 7.1, there clearly existed an L1 data cache configuration below which the miss rate drastically increased. For most of these programs it lies somewhere around the 4KB 2-way enabled configuration.

Benchmark	Sim num instructions	L1 dcache enables	L1 dcache disables	L1 dcache resets
164.gzip	10372169311	165	631	226
175.vpr	26313787308	10347	49649	19199
177.mesa	120864748	87	3027	1009
179.art	2233642507	0	16	6
181.mcf	601286542	0	27	11
183.quake	715895043	64	1437	468
188.ammmp	1225966383	2	589	119
197.parser	2968376902	52	478	217
adpcm	6532795	0	3	0
mpeg2	1133799259	748	2585	736
epic	52742246	13	33	6

Table 7.2 : Reconfigurable L1 Data Cache via Hardware Sampling: Instruction Counts and Reconfiguration Statistics

Why then did the hardware performance monitoring mechanism not settle on or about this point? The answer to this question actually has nothing to do with the 4KB 2-way enabled configuration, but rather the left most point on each plot in which the L1 data cache is fully enabled. Remember that in Chapter 4, the thresholds at which the L1 data cache decides to enable or disable a given way are statically set. They can not adapt during program runtime, and are specifically set to disable a given L1 data cache way when the L1 data cache miss rate drops below 3%. For each of the benchmarks that spend the majority of runtime with the L1 data cache fully enabled, it can be seen that L1 data cache miss rate lies well above 3%. This means that even with the L1 data cache fully enabled during the program's execution, there were L1 data cache misses occurring frequently enough that the threshold to disable an L1 cache way was infrequently crossed. To the hardware, it appeared that the miss rate was high enough to indicate that either the working data set exceeded the maximum possible size the L1 data cache was configured to, or that the working data set exhibited very little temporal or spatial locality. Looking at Table 7.3, some of the benchmarks for which the L1 data cache was fully enabled

Benchmark	8KB static L1 miss rate	Hw Reconfig L1 miss rate	% diff for Hw Reconfig
164.gzip	0.0589	0.0620	5.26
175.vpr	0.0203	0.0259	27.58
177.mesa	0.0029	0.0106	265.52
179.art	0.2476	0.2476	0.00
181.mcf	0.2417	0.2419	0.08
183.quake	0.0203	0.0280	3.79
188.ammp	0.3012	0.3034	0.73
197.parser	0.0470	0.0477	1.49
adpcm	0.0006	0.0188	3033.33
mpeg2	0.0017	0.0223	35.29
epic	0.0567	0.0626	10.40

Table 7.3 : Reconfigurable L1 Data Cache via Hardware Sampling: Relative Change in L1 Data Cache Miss Rates

during the majority of runtime exhibited a significant amount of cache reconfiguring during program execution. It is suspected that such programs exhibited periods of a few hundred thousand instructions during which large amounts of temporal locality were exploited. During such a period, the L1 miss rate would drop due to the large amounts of data reuse. This, in turn, would cause the L1 data cache to disable a way. At this point, either the data for which temporal reuse was being exploited was flushed from the L1 data cache when the way was disabled, or the current working data set changed resulting in cold misses. In either case the L1 data cache either reset to full size or the way was enabled again.

If, in fact, there existed an optimal L1 data cache configuration to operate at for the benchmarks which spent the majority of execution time with the L1 data cache fully enabled, the hardware reconfigurable scheme failed to find it. This is not to say that such a solution is not possible in hardware alone. If our target machine could track the relative L1 data cache miss rates throughout execution for each possible L1 data cache configuration, and dynamically adjust the thresholds at which L1 data cache ways are enabled or disabled, then optimal performance could be achieved.

This, of course, assumes that the working data set of the program remained live for a great enough period of time for the hardware to detect the temporal and spatial reuse occurring in the L1 data cache. Such a technology would require more transistors to track phase shifts in program behavior etc., and was beyond the scope of this work. It still, however, provides an interesting subject for future research as the average cache miss rates of all programs are clearly not the same.

For the remaining benchmarks simulated, a significant portion of the program runtime was spent with various portions of the L1 data cache disabled. The benchmarks 175.vpr, 177.mesa, 183.quake and Epic, only spent about 50% of the total program runtime with the L1 data cache enabled, while Adpcm spent less than 10% of total program runtime with the full L1 data cache enabled. Why did such behavior occur in these benchmarks, and not in the benchmarks described in the previous paragraphs? Adpcm, for example, spends almost 85% of the total runtime with only one L1 data cache way enabled, and almost 10% with four L1 data cache ways enabled. Additionally, the only L1 data cache reconfiguring that occurs is three disables during program runtime, as is shown in Table 7.3. The reason for this is that the working set of Adpcm was small enough that it fit into the L1 data cache when only one cache way was enabled. The three cache way disables that occurred during execution occurred at program startup, after the number of cold misses occurring in the L1 data cache settled down and reached steady state. The reason that this settling down period accounted for almost 10% of the runtime is that Adpcm executes very few instructions when compared to the other benchmarks shown. It takes a few hundred thousand instructions for the initial cold misses to occur in the L1 data cache, and then for the L1 data cache to continue disabling cache ways down to the 2KB direct-mapped state.

Looking at 175.vpr in Table 7.3, more than 50% of the total runtime was spent with the L1 data cache fully enabled. Just over 35% was spent with between two and three L1 data cache ways enabled, while the remaining runtime consisted of having

only one L1 data cache way enabled. Yet looking at the data presented in Figure 7.1, we see that when the L1 data cache is statically configured to three ways enabled, the corresponding miss rate lies just below 3%, while when the L1 data cache is configured to two ways enabled the corresponding miss rate is slightly above 3%. Intuitively, one might think that based on this data, allowing the L1 data cache size to be reconfigured at runtime would result in the L1 data cache size hovering between the two way enabled and three way enabled configurations. Such an assumption, however, would be naive. If we look at the average L1 data cache miss rate for 175.vpr when the cache is dynamically reconfigured at runtime, it indeed lies somewhere just below 3%. The reason that the L1 data cache is not configured to either two or three ways enabled during the majority of program runtime lies in the fact that the data presented in Figure 7.1 is an average over the entire program's execution. Looking at the number of times the L1 data cache is reconfigured, as presented in Table 7.3, it is clear that the L1 data cache miss performance is not uniform throughout execution. 175.vpr contains many small loops which execute for relatively small number of iterations, after which control flow moves on. It is suspected that when control flow enters a new loop nest, cold misses occur resulting in the L1 data cache resetting to fully enabled. This is because the hardware assumes the working data set of the program has changed, and no information about the new working data set is known. Once execution continues in the loop nest, and the cold misses settle down, the L1 data cache miss rate drops. Once below the 3% mark, an L1 data cache way is disabled. This continues until control flow exits the loop nest and enters another nest at which point the process starts over again. The reason such a small amount of program runtime is spent with only one L1 data cache way enabled is that many of the loop nests do not execute enough instructions before control flow exits. With only a small instruction stream exhibiting low L1 data cache miss rates, not enough insight can be gathered by the hardware to warrant disabling all but one of the L1 data cache ways. Perhaps if larger input data sets were used for the simulation, the iteration space of

such loop nests would be larger resulting in a larger percentage of program runtime during which only one or two L1 data cache ways were enabled.

The question then becomes, how does the performance and configuration of the hardware reconfigurable L1 data cache relate to power consumption. As was described in Chapter 4, the hardware performance monitoring mechanism attempts to minimize the total amount of L1 data cache enabled at any given time. By doing this, power will be conserved on each L1 data cache access by not accessing multiple L1 data cache ways in parallel. Additionally, due to the gated V_{dd} design proposed by Falsafi et. al, the amount of leakage current present in the disabled L1 data cache cells should be significantly reduced [19] [27]. It is important that the L1 data cache size be minimized only to the point whereby further decreasing the size would result in an increase in the L1 data cache miss rate. The reasoning for this is that although power will be conserved in accesses to the L1 data cache, these savings will be offset by subsequent accesses to the much larger unified L2 cache.

Table 7.3 shows the relative increase in L1 data cache miss rates for the hardware reconfigurable mechanism versus running with all available L1 data cache enabled. Those benchmarks that left the L1 data cache fully enabled for the majority of a program's execution show little change in the L1 data cache miss rates, as would be expected. For those programs which disabled portions of the L1 data cache during execution, the relative increases in L1 data cache miss rates vary greatly. During its execution, 183.quake left the entire L1 data cache enabled for only 58.46% of its total runtime. The rest of the runtime was evenly distributed between having the one, two, and three L1 data cache ways enabled at any given time. Looking at the relative increase in L1 data cache miss rates in Table 7.3, we see that reconfiguring the L1 data cache size at runtime only increased 183.quake's overall L1 data cache miss rate by 3.79%. Clearly, having a number of L1 data cache ways disabled for over 40% of the runtime will save power consumed on L1 data cache accesses. Additionally, since the increase in L1 data cache miss rate is only 3.79%, the number of L1 accesses

which now result in misses and subsequent unified L2 cache accesses will be fairly low. In a case like this, it is highly likely that the additional power consumed by the unified L2 cache will be offset by the power savings gained in the L1 data cache.

The relative increase in miss rate associated with reconfiguring the L1 data cache at runtime is not always as subtle as it was with 183.quake. Both 177.mesa and Adpcm spend significant amounts of runtime with portions of the L1 data cache disabled. 177.mesa uses the full L1 data cache for approximately 50% of its runtime, while Adpcm uses the full data cache for only 10% of its runtime; the rest of the runtime using only one way in the L1 data cache. Yet the increase in miss rate when the L1 data cache is reconfigured at runtime is very high. For 177.mesa, the increase in L1 data cache miss rate is over 200%, while in Adpcm it is over 3000%. At first glance, one might suspect that such a drastic increase in L1 data cache miss rate would result in increased overall power consumption. This, however, may not be the case for these benchmarks. Notice that with the L1 data cache always fully enabled, the miss rate for 177.mesa is on the order of 0.29%. For Adpcm, it's even lower at 0.06%. Although the relative increase in miss rate is quite large when the L1 data cache is reconfigured dynamically, the overall L1 data cache miss rate still stays well below 2% for both of these benchmarks. This is to say that, although there is a marked increase, the resulting L1 data cache miss rate may still be low enough as to not incur enough power consumption at the unified L2 level so as to offset the power savings in the L1 level. This is most certainly the case with Adpcm, which spends 90% of the runtime with only one L1 data cache way enabled. It may very well be the case with 177.mesa as well.

7.4 Reconfiguring the Cache Via Compiler Time Analysis of Loop Nests

In this section, the results of analyzing the L1 data cache performance of loop nests at compile time is presented. As was described in Chapter 5, the compiler finds

Benchmark	Total Loops	Anal Loops	% Anal Loop Runtime
164.zip	194	14	0.0
175.vpr	134	9	0.3
177.mesa	934	11	0.1
179.art	75	0	0.0
181.mcf	45	1	0.0
183.equake	90	16	0.6
188.ammp	257	13	0.8
197.parser	531	1	0.0
adpcm	7	0	0.0
mpeg2	168	16	0.0
epic	163	18	1.7

Table 7.4 : Analyzable Loop Nests in SPEC CPU2000 and Mediabench Benchmarks

perfectly nested loops, or loop nests with no control flow in them which meet the previously described criteria for analyzability. For each of these loops, the compiler forms and solves a set of equations that model potential cold, capacity, and conflict misses in the L1 data cache. Using this information on potential cold, capacity, and conflict misses, the compiler determines the amount of L1 data cache to enable during the loop's execution. The compiler selects the minimal amount of L1 data cache to enable such that the maximum number of potential L1 data cache misses are avoided, as was described in Section 5.5. By doing this, costly accesses to the much larger and slower unified L2 cache can be avoided. At the same time, power will be saved by not accessing as many L1 data cache ways in parallel. Using the compiler hints in combination with the hardware performance monitoring mechanism discussed in Section 7.2, it is hoped that better L1 data cache performance will be realized. For regions of the code where the compiler can not determine cache performance, the hardware performance monitoring mechanism will adapt. On the other hand, for loop nests where the compiler can predict performance, the L1 data cache can be configured at the start of loop nest execution without wasting valuable power adapting to future program behavior.

Table 7.4 lists the total number of loops found in each benchmark, as well as the number of loops that met the criteria for analyzability described in Chapter 5. The final column in Table 7.4 lists the amount of program runtime the compiler analyzable loops made up, as was determined using the *gprof* code profiling utility. Although some of the benchmarks analyzed contained more analyzable loops and loop nests than others, the total runtime that these loops made up was discouragingly low. Even if the reconfigurable L1 data cache could be set to the smallest configuration during execution of every analyzable loop, the power savings at the L1 data cache level would be negligible. Clearly not enough program runtime could be covered to make a difference in such programs. Why did the compiler fail analyze portions of the input program which dominated runtime? The major reason was that the criteria for compiler analyzability loop nests were too strict. For many loops, determining loop bounds at compile time was a problem. This was more prevalent in the SPEC CPU2000 benchmarks. Another problem quite common in the SPEC CPU2000 benchmarks was the control flow in loop nests. The compiler could not ensure predictable control flow within the loop nest, and thus failed to analyze it. Finally, many of the benchmarks had pointers through which data structures were accessed. The compiler could not disambiguate what memory was being referenced through the pointer, and thus could not accurately analyze the loop.

Looking at the third column of Table 7.4, it can be seen that a number of loops in each program did meet the criteria for analyzability. Why then didn't these loops account for more program runtime? Many of the loops that met the criteria for analyzability were used to initialize fields in data structures. Quite often, analyzable loops did nothing more than set arrays to zero, or initialize bit field. In addition to this, many analyzable regions were not loop nests but rather single loops. These single loops rarely exhibited any temporal reuse in the L1 data cache. For almost all loops analyzed, the compiler decided to disable as much L1 data cache as possible. There was no temporal reuse to be exploited, and so few conflicts occurred that disabling as

much L1 data cache as possible didn't increase the L1 data cache miss rate.

In light of the data presented in Table 7.4, it is clear that in order to predict L1 data cache performance of loop nests in real programs, a more robust analyzer is needed. The question then becomes, what do loop nests which dominate the program runtime and exhibit interesting cache behavior look like? Are such loop nests analyzable at compile time? What are the reuse patterns of memory references in such loops, and are they highly sensitive to L1 data cache associativity? Does the majority of temporal reuse in programs occur within loop bounds? In order to find these answers, a custom L1 cache simulator was designed to analyze the L1 data cache performance of all loops in a given benchmark. More specifically, the reuse information for each unique memory reference within a loop nest was analyzed. By finding loop nests that contain memory references with reuse distances that can be realized in the reconfigurable L1 data cache, better insights to the requirements of our compiler technology can be gained. Details of this experiment are described in Chapter 8.

Chapter 8

A Study of Loop Nest Data Reuses for SPEC CPU2000 and Mediabench Programs

Upon examining the total amount of program runtime analyzable by our compiler pass, as was described in Chapter 7, it became apparent that additional investigation into program loop nest behavior was necessary. Clearly our compile time requirements for loop nest analyzability were too strict. Although a number of loops in the programs analyzed met our restrictions, they accounted for very little of the total program runtime. Did there exist loop nests within the programs we analyzed which accounted for a significant portion of the program runtime, and also contained memory references with non-zero reuse distances? If there were loop nests containing memory references with non-zero reuse distances, were the reuse distances of these memory references short enough such that they could be realized by one of the associativities our reconfigurable L1 data cache could be set to? If so, the compiler might have a chance at finding an ideal L1 data cache configuration for such loop nests which would reduce the overall processor power consumption during loop nest execution.

8.1 Loop Nest Cache Analysis Framework

By analyzing each loop in a given input program, we hope to find memory references with reuse distances which can be realized in the target machine's reconfigurable L1 data cache. In addition to this, such loop nests should have a relatively simple code shape which can be analyzed by the compiler. If such loop nests exist, then it may be possible for the compiler to predict an optimal L1 data cache configuration during

loop nest execution as described in Section 5.5.

In order to analyze the cache performance of every loop in the input program, a memory trace was needed for each program to be analyzed. This was achieved using a modified version of SimpleScalar’s `sim-profile` tool [5]. Special care was taken to ensure that memory traces were only generated for code contained within loop bounds, such as *for* loops, *while* loops, and *do-while* loops. The first step in achieving this was to annotate the beginning and end of each loop in the original source code, as well as giving each loop in the program a unique identification number. Once the loops were annotated, the SimpleScalar PISA executable was generated with the modified version of the SimpleScalar PISA compiler as described in Chapter 6. SimpleScalar’s `sim-profile` tool was modified to dump a memory traces whenever control flow of the source program entered a loop, as was identified by the annotated instructions inserted prior to compilation. Once the trace of loop nest’s memory references was generated for the input program, each individual memory reference’s reuse behavior was analyzed.

A custom L1 data cache simulator was built to analyze the reuse distances of memory references inside loop nest. The simulator modeled the contents of our L1 data cache in response to memory references inside the loop nest, but the associativity of our simulated L1 data cache was effectively infinite. By doing this, we were able to track the number of conflicts to a given cache set between successive reuses of a given memory reference. Figure 8.1 and Section 8.2 illustrate the algorithm used in analyzing each loop’s memory reference reuse information.

8.2 Loop Nest Cache Analysis Algorithm

As is depicted in the algorithm shown in Figure 8.1, for each loop in the program to be analyzed, a set of memory lines is allocated for each L1 data cache set on the target machine. We herein refer to each set of memory lines as a *Cache_Way_Conflict_Set*. Each set in our simulated L1 data cache has a corresponding *Cache_Way_Conflict_Set*

Input: Trace of memory references that occur during execution of a given loop.

Output: Distribution of reuse distances that occur for each distinct address referenced inside a given loop.

```

For each loop,  $L_i$ , control flow passes through in the input program {
  For each cache set on target machine,  $C$  {
     $Cache\_Way\_Conflict\_Set[C] = \emptyset$ ;
  }

  For each memory reference  $Mem\_Ref_{L_i}$  in loop  $L_i$  {
    Load memory line  $Mem\_Line_{L_i}$ , where  $Mem\_Ref_{L_i} \in Mem\_Line_{L_i}$  into cache set  $C$ ;
    For each memory line  $M \in Cache\_Way\_Conflict\_Set[C]$  {
      If  $Mem\_Line_{L_i} = M$  {
        Record conflict count of  $Mem\_Ref_{L_i}$ ;
         $Mem\_Ref_{L_i}$ 's conflict count = 0;
      }
      Else {
        For each reference  $R \in Mem\_Line_{L_i}$  {
           $R$ 's conflict count++;
        }
      }
    }
     $Cache\_Way\_Conflict\_Set[C] \cup Mem\_Line_{L_i}$ ;
  }
}

```

Figure 8.1 : Algorithm to Analyze Loop Nest Data Reuses

which contains all the memory lines accessed within the loop nest that map to a given cache set. Using the *Cache_Way_Conflict_Set*, we can then determine number of conflicts that occur to a given cache set between successive data reuses. This then tell us the amount of L1 data cache associativity necessary in order to realize this reuse in L1 data cache. At the start of analyzing a trace of memory references for a given loop's execution, each set of memory lines $S \in Cache_Way_Conflict_Set$ is initialized to the null set, \emptyset . Once each *Cache_Way_Conflict_Set* is initialized to \emptyset , processing of the loop's memory references begins. For each memory reference

$Mem_Ref_{L_i}$ accessed within the loop L_i , the memory line $Mem_Line_{L_i}$ containing the memory reference $Mem_Ref_{L_i}$ is loaded into the appropriate L1 data cache set C , as is done in a normal data cache.

Once the memory line $Mem_Line_{L_i}$ is loaded into the appropriate L1 data cache set C , conflict information must be updated for every memory reference accessed so far which lied in a memory line that mapped to the L1 data cache set C . In order to do this, we iterate over all the memory lines $M \in Cache_Way_Conflict_Set[C]$. If the memory line $Mem_Line_{L_i}$ containing the current memory address being processed is identical to the memory line $M \in Cache_Way_Conflict_Set[C]$, then a reuse has occurred. In this case, the current memory reference $Mem_Ref_{L_i}$ has a reuse distance recorded which is equal to the number of cache line conflicts that have occurred to that memory line containing $Mem_Ref_{L_i}$ since it was last referenced. Once the reuse distance is recorded, the conflict count for $Mem_Ref_{L_i}$ is reset to zero. If, on the other hand, the memory line $Mem_Line_{L_i}$ containing the address currently being processed is not the same as the memory line M contained in $Mem_Lines_Accessed[C]$, then an L1 data cache conflict has occurred for all memory references whose address is contained in the memory line M . In this case, all memory references already seen in the input stream which lie within memory line M have their conflict counts incremented by one. This represents an increase in the reuse distance between the last reference to this memory address, and future reuse in the instruction stream.

8.2.1 Caveats to Loop Nest Reuse Analysis

For all SPEC CPU2000 benchmarks using the full input data sets, trace analysis time greatly exceeded 800 hours on a dedicated AMD Athlon 1800+ MP machine. As a result, the reduced input data sets developed by Kleinosowski were used for the SPEC CPU2000 benchmarks [14]. Still, trace analysis computation times were quite unreasonable. For an input data set which resulted in the execution of approximately 5 billion instructions, trace analysis times were still on the order of 600+ hours on

a dedicated AMD Athlon 1800+ MP machine for many of the floating point benchmarks. The reason for this was the large number of deeply nested loops containing many memory references. With a large input data set, the number of memory references *live* within a given loop nest grew quite large. This resulted in an extremely large search space to traverse when determining conflict distances of various memory references occurring within the loop nest. As a result, for many of the SPEC CPU2000 traces, the small input data sets developed by Kleinosowski [14] were used, resulting in the execution of between 500 million and 1 billion instructions. Due to the reduced size of the input data sets used, the overall L1 data cache behavior of the SPEC CPU2000 benchmarks changed from what was seen using the original SPEC CPU2000 input data sets. For many of the applications, the average size of the working set was significantly smaller, resulting in a very high L1 data cache hit rate [6] [14] [17]. Because of this, when analyzing the traces generated for each of the benchmarks, we considered an L1 data cache with 32 byte line size as before, but only 64 cache sets as compared to the original L1 data cache configuration described in 8.1. For the given number of L1 data cache ways which can be enabled on the target machine, this maps to the possible configurations of a 2KB direct mapped, 4KB two-way associative, 6KB three-way associative, and 8KB four-way associative L1 data cache.

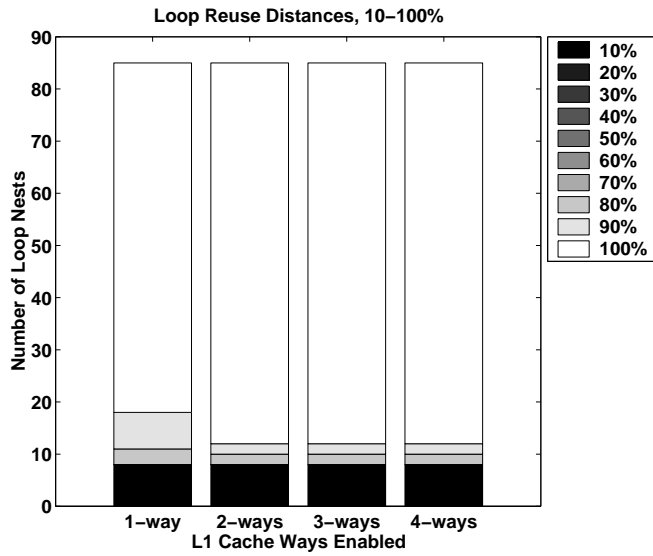
8.3 SPEC CPU2000 and Mediabench Loop Nest Data Reuse Results

Figures 8.2 – 8.7 show the results obtained from modeling data reuses within loop nests of the SPEC CPU2000 and Mediabench benchmarks. Each of the figures shows the distribution of reuses realized per loop versus the L1 data cache associativity. For example, looking at the data for 183.equake in Figure 8.3, the first bar in the graph shows that 14 of the 85 loops in the program have 10% or less of their data reuses realized with only one L1 data cache way enabled. Similarly, 28 of the 85 loops in

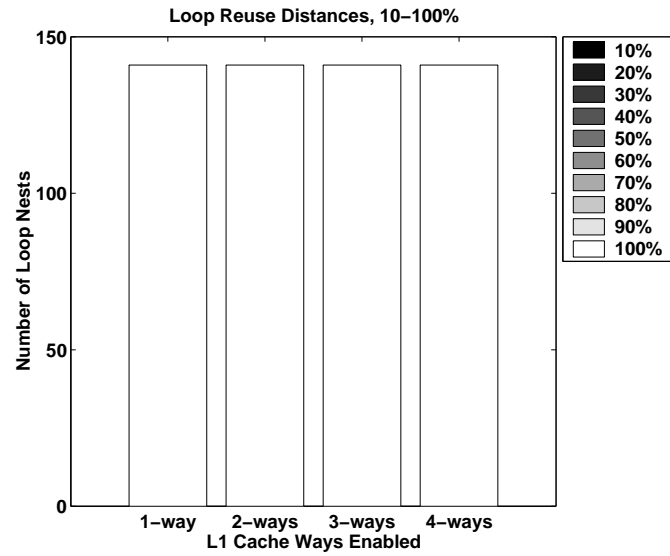
the program have 90% or fewer of their total data reuses realized with only one L1 data cache way enabled. The remaining bars in this figure illustrate the same data with two, three, and four L1 data cache ways enabled. Figure 8.4 shows the same data for the Mediabench benchmarks simulated. Figures 8.5, 8.6 and 8.7 show the same data as Figures 8.2, 8.3 and 8.4 respectively, but with a distribution of reuses from 90%–100% rather than 10%–100%.

Looking at the data for Mpeg2 in Figure 8.4, the region in each bar labeled 10% shows that there are 40 or so loops in the program which are dominated by data reuses that have four or more conflicts between successive reuses. That is to say, even with our reconfigurable data cache fully enabled, the majority of reuses in these loops will still result in misses in the L1 data cache. On the contrary, the region labeled 100% shows that 45 of the 130 total loops in the program have between 90% and 100% of their data reuses realized with only one way enabled in the reconfigurable L1 data cache. The most interesting region in these figures is the distribution of loops lying between 20% and 90%. Again referring to Mpeg2 in Figure 8.4, as the number of ways enabled in the reconfigurable L1 data cache is increased, the number of loops in each distribution between 20% and 90% decreases. What this is really showing is that as L1 data cache associativity increases, there are a significant number of loops in the program that have an increasing number of data reuses realized. Loops that may have been grouped in the 20% reuses realized region with only one L1 data cache way enabled, now have a higher percentage of reuses realized. This accounts for the downward slope in distributions as the L1 data cache associativity is increased.

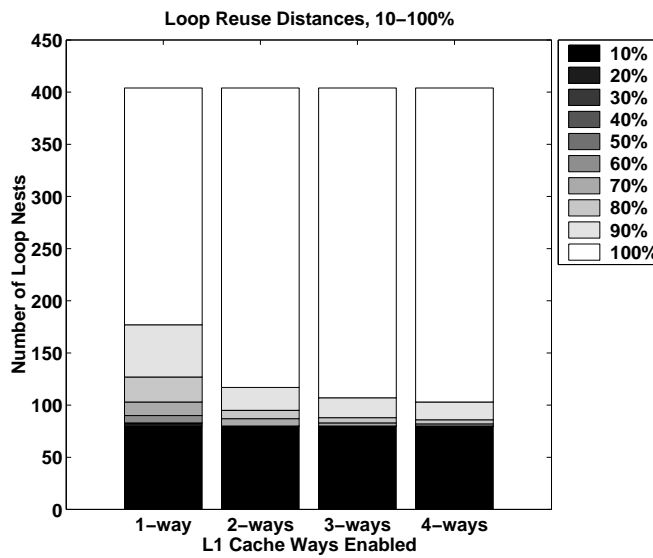
Looking at Figures 8.2– 8.7, it is clear that the majority of the benchmarks simulated have a significant number of loops dominated by data reuses that are sensitive to L1 data cache associativity. In addition to this, there are a significant number of loops which are dominated by data reuses independent of L1 data cache associativity. For these loops, either the majority of data reuses have zero L1 data cache conflicts between successive reuses, or the number of conflicts between successive reuses is



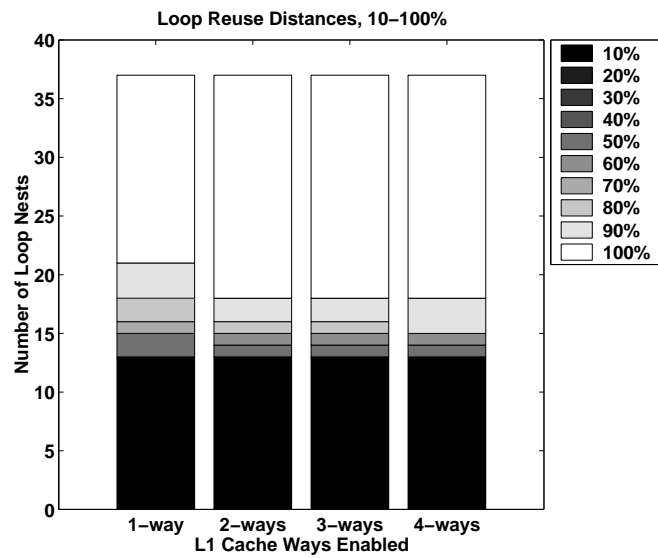
(a) 164.zip



(b) 175.vpr

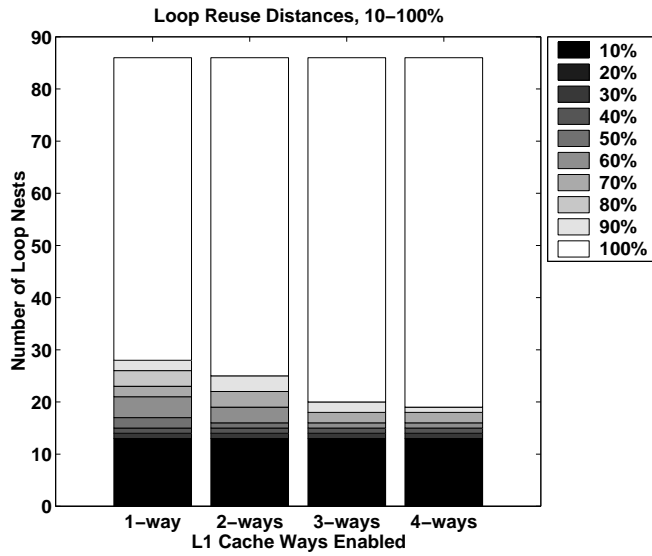


(c) 197.parser

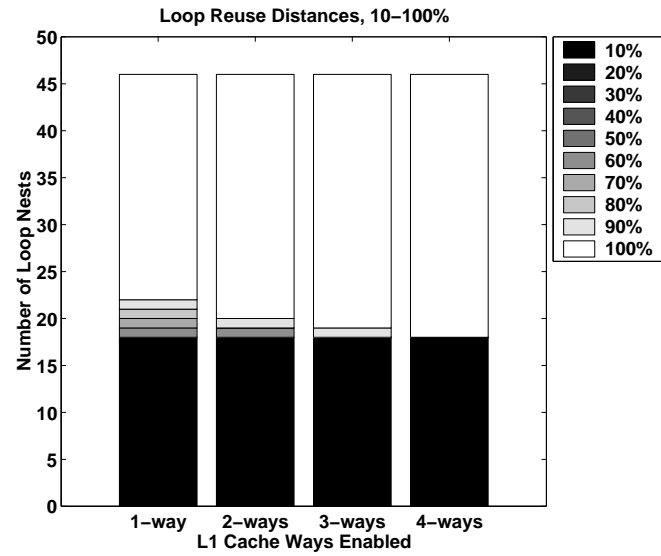


(d) 181.mcf

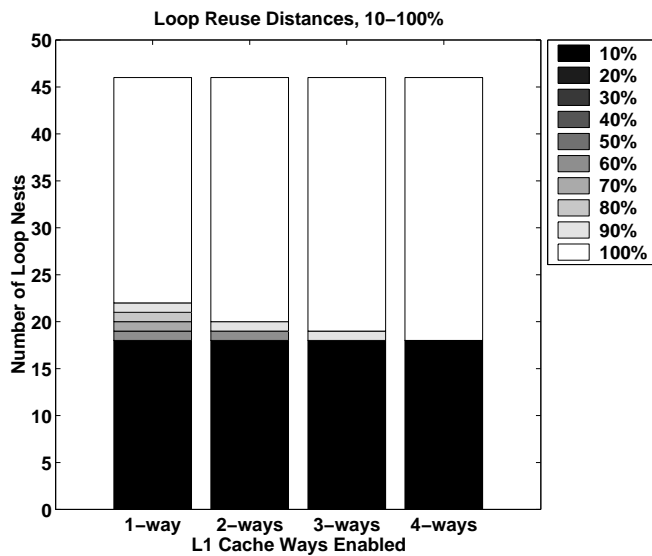
Figure 8.2 : Distribution of Reuses Realized Per Loops Nest vs. L1 Data Cache Associativity, SPEC CPU2000 Integer Programs, 10%–100%



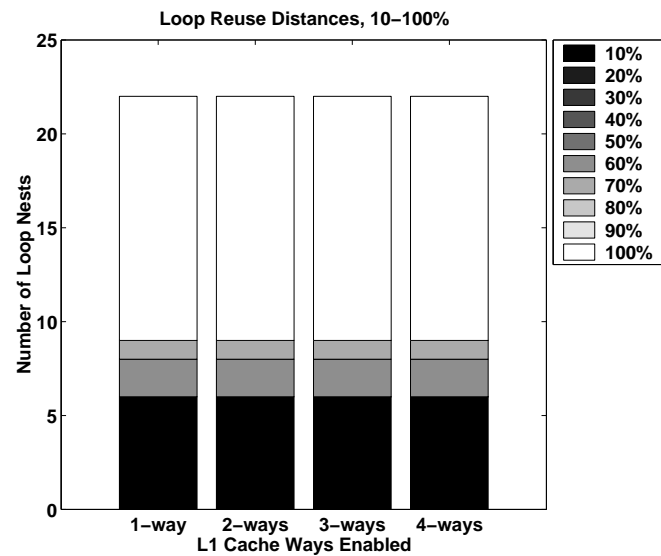
(a) 183.quake



(b) 179.art

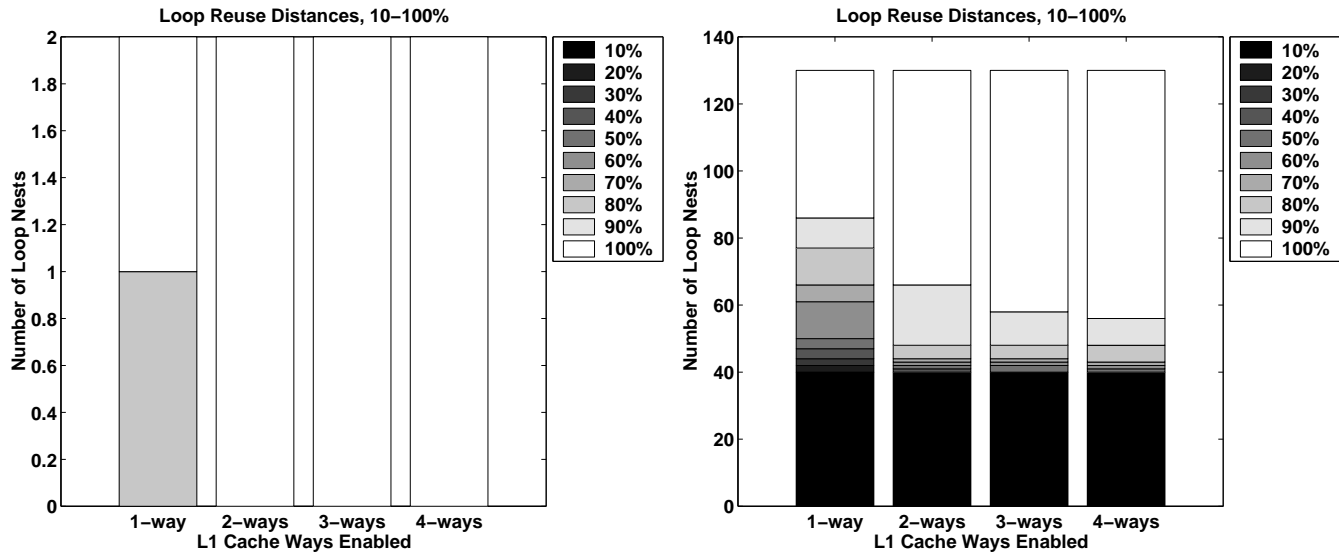


(c) 177.mesa



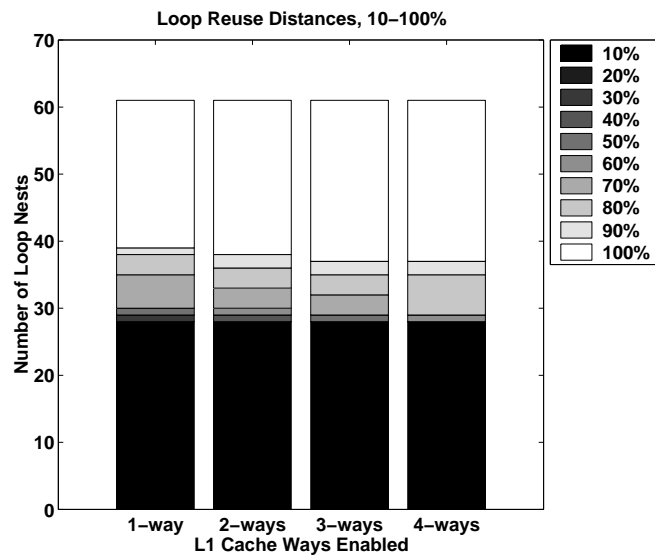
(d) 188.ammp

Figure 8.3 : Distribution of Reuses Realized Per Loops Nest vs. L1 Data Cache Associativity, SPEC CPU2000 Floating Point, 10%–100%



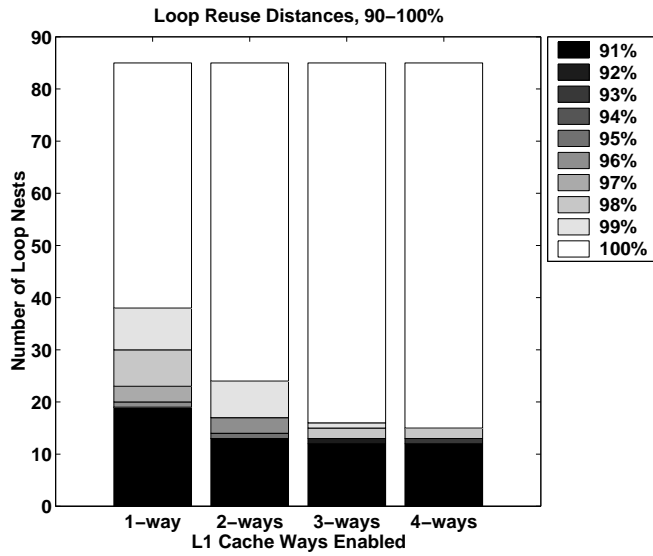
(a) Adpcm

(b) Mpeg2 (encode)

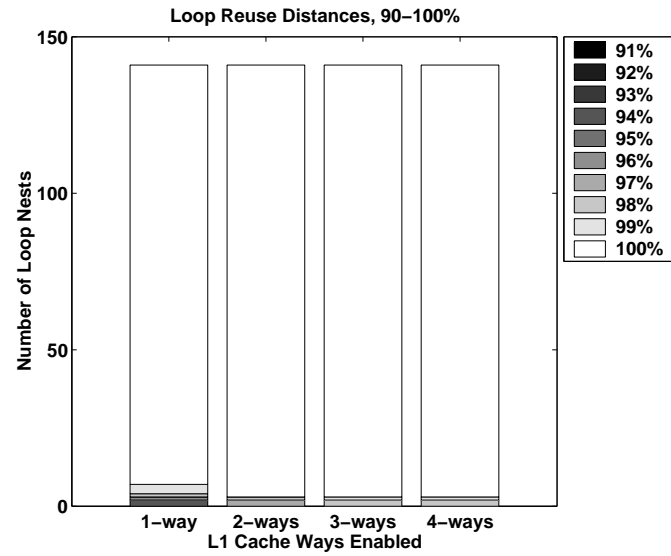


(c) Epic

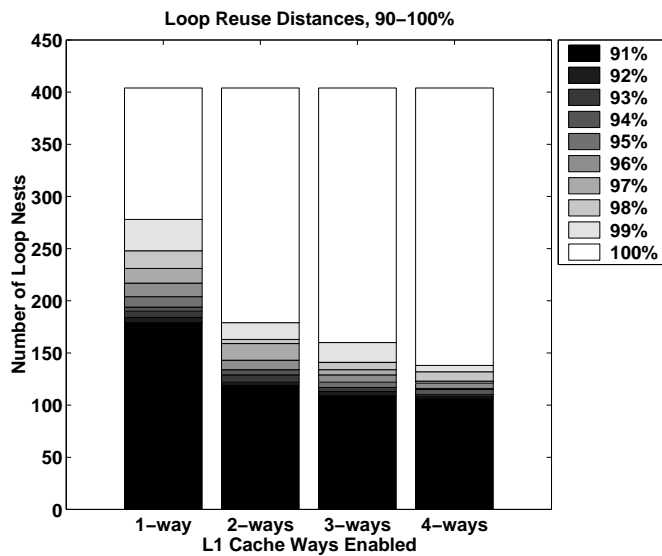
Figure 8.4 : Distribution of Reuses Realized Per Loops Nest vs. L1 Data Cache Associativity, Mediabench Programs, 10%–100%



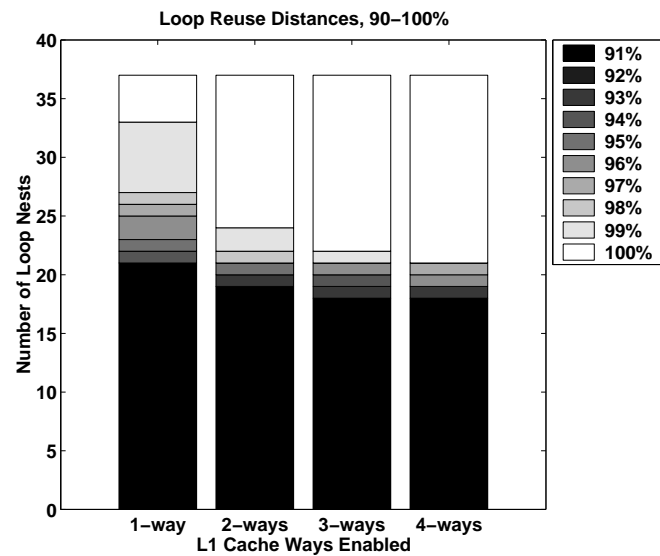
(a) 164.zip



(b) 175.vpr

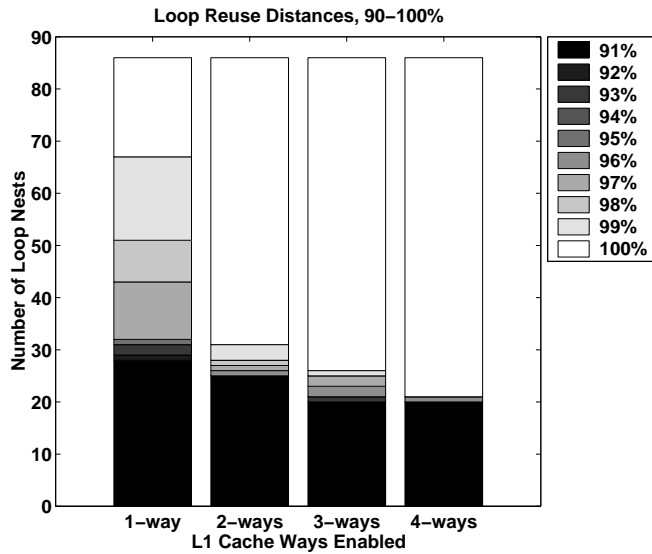


(c) 197.parser

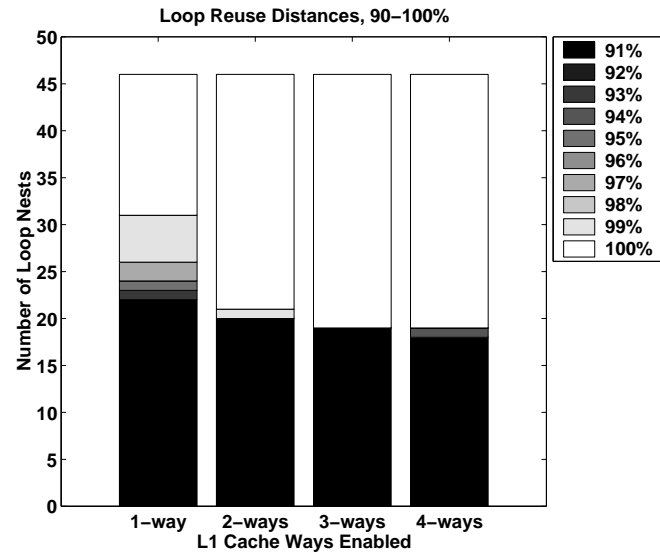


(d) 181.mcf

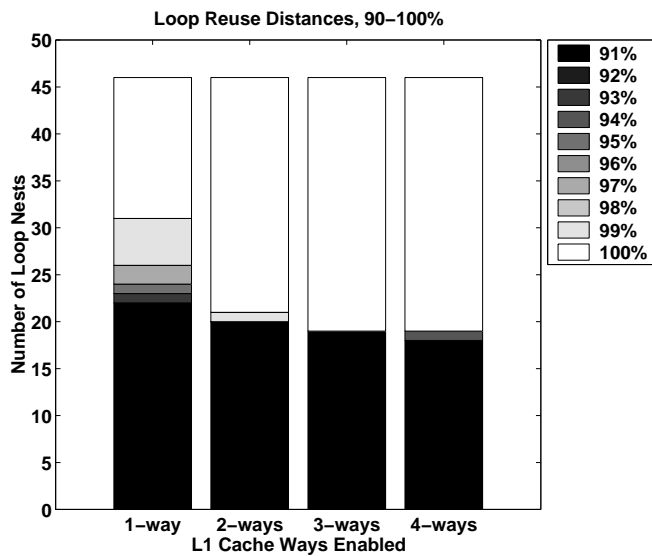
Figure 8.5 : Distribution of Reuses Realized Per Loops Nest vs. L1 Data Cache Associativity, SPEC CPU2000 Integer Programs, 90%–100%



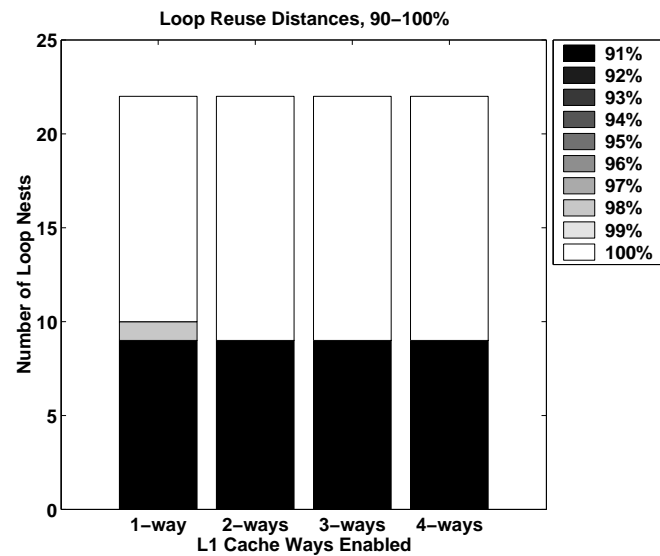
(a) 183.equake



(b) 179.art

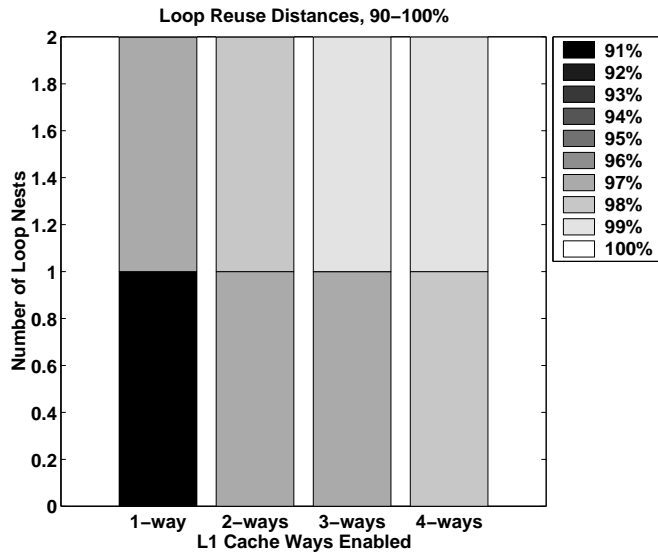


(c) 177.mesa

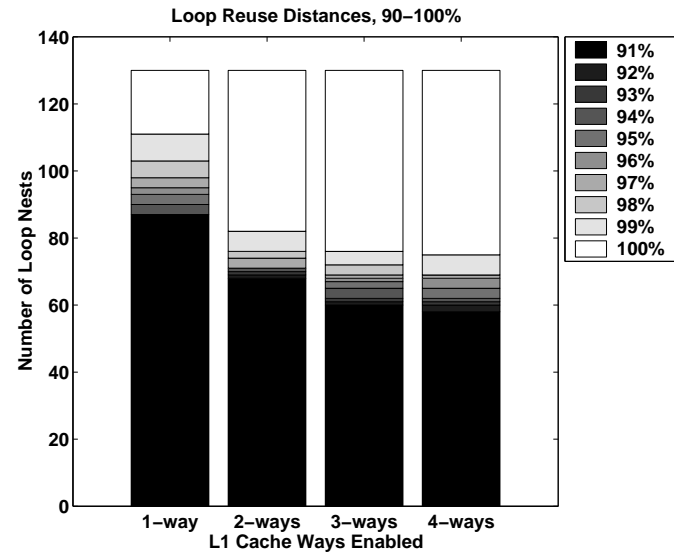


(d) 188.ammp

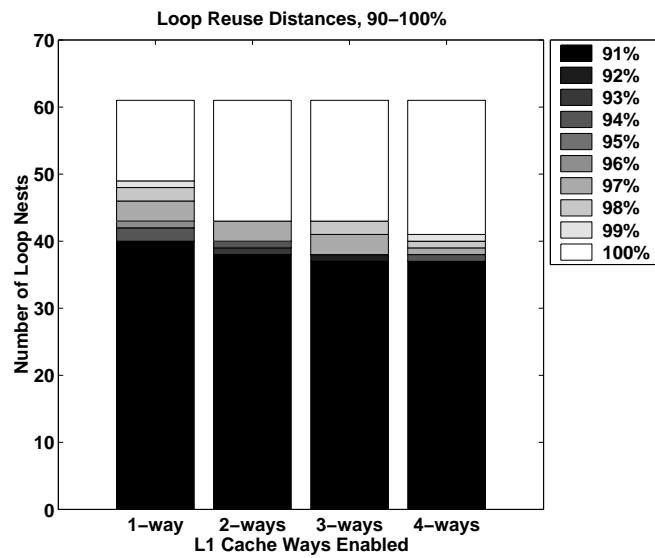
Figure 8.6 : Distribution of Reuses Realized Per Loops Nest vs. L1 Data Cache Associativity, SPEC CPU2000 Floating Point Programs, 90%–100%



(a) Adpcm



(b) Mpeg2 (encode)



(c) Epic

Figure 8.7 : Distribution of Reuses Realized Per Loops Nest vs. L1 Data Cache Associativity, Mediabench Programs, 90%–100%

large enough that the reuse cannot be realized in a four way associative L1 data cache. In either case, it is obvious that L1 data cache behavior is not uniform across all loops in these benchmarks. It is important to note, however, that the majority of these benchmarks exhibit a point of diminishing returns. Beyond two or three ways enabled in the reconfigurable L1 data cache, most benchmarks show little increase in percentage of data reuses realized.

Why then shouldn't we just use hardware performance monitoring similar to that described in Chapter 4, and avoid compiler analysis altogether? The primary reason is that the hardware must adapt to L1 cache behavior, whereas the compiler can reconfigure the L1 data cache in advance. Many of these benchmarks consist of loops which execute for a few hundred thousand or million instructions, such as `175.vpr`. The majority of benchmarks also contain a significant number of loops which are dominated by data reuses that are independent of L1 data cache associativity. This is because either there are zero L1 data cache conflicts between successive reuses of data, or the number of conflicts between successive reuses is high enough that reuse can not be realized by our reconfigurable L1 data cache. For these loops, the L1 data cache can be fully disabled without deteriorating performance greatly, and thus power savings can be achieved. As we saw in Chapter 7, the hardware performance monitoring mechanism rarely operated in the fully disabled L1 data cache configuration. This is often because the hardware needed to adapt to the L1 data cache behavior, and could not reconfigure the L1 data cache from fully enabled to fully disabled in a single step. It can take as long as 300,000 instructions before the hardware performance monitoring can downsize the L1 data cache. When programs consist of loops which iterate over a working set for only a few hundred thousand instructions, the hardware will rarely have a chance to adapt accordingly. Considering the number of loops in these programs which maintain L1 data cache performance with the reconfigurable L1 data cache fully disabled, it becomes clear that there are improvements which can be gained from compile time analysis. Considering the number of loops in these

programs which are sensitive to L1 data cache associativity, always setting the L1 data cache to fully disabled is not a viable alternative either. A number of loops in each benchmark will achieve the maximum number of data reuses with only two or three ways enabled in the L1 data cache. Enabling more ways in the reconfigurable L1 data cache will not increase the L1 data cache hit rate in these loops nest, and will just consume more power in the cache hierarchy. Although the hardware performance monitoring can adapt to this type of program behavior, there is a significant amount of error which can be corrected for using compile time information.

Chapter 9

Conclusions and Future Work

This body of work shows that there are opportunities to reduce power consumption in the data cache hierarchy using a reconfigurable L1 data cache. The hardware performance monitoring mechanism presented in Chapter 4 performed better on some applications than others, due to its adaptive nature. An interesting avenue of future research lies in dynamically adjusting the thresholds at which the L1 data cache can be resized. Data cache miss rates are not uniform across all applications. The ability to adjust the thresholds at which the L1 data cache is reconfigured on a per application basis would most likely provide improved results.

In Chapter 7, it was shown that it is difficult to predict L1 data cache performance of loop nests in real programs. A large amount of compile time information is necessary to build the system of cache miss equations as described in Chapter 5. Many real programs contain loop nests that are not written in a manner easily analyzable at compile time. Quite often, lack of information on loop bounds, and control flow within loop nests made loops unanalyzable by our algorithm. This is not to say that compile time analysis of L1 data cache performance can not provide additional savings in L1 data cache power consumption over adaptive hardware mechanisms. In the case of control flow within loop nests, it may be the case that modeling only the dominant control flow path provides accurate enough insight on L1 data cache performance. Additionally, loop bounds that could not be determined at compile time could be solved parametrically and the L1 data cache could be configured based on the runtime value of a loop's bounds. Techniques such as profiling could provide insight on values which are otherwise indeterminate at compile time. Using this informa-

tion, the compiler may then be able to provide improvements in L1 data cache power consumption by avoiding the inefficient nature of adaptive hardware techniques.

Finally, it was shown that there are a significant number of loops in the SPEC CPU2000 and Mediabench benchmarks that contain data reuses sensitive to L1 data cache associativity. Many loops can achieve maximal L1 data cache performance with a direct mapped L1 data cache, as the data referenced within them have zero L1 data cache conflicts between successive reuses, or a number of L1 data cache conflicts large enough that the reuse cannot be realized in a four way associative L1 data cache. There are also a number of loops in these benchmarks which achieve maximal L1 data cache performance with a two or three way set associative L1 data cache. Increasing the associativity of the L1 data cache provides no increase in the L1 data cache hit rate of these loops. These results strengthen the argument for analyzing cache performance of loop nests at compile time in order to decrease power consumption of the data cache hierarchy.

Bibliography

- [1] Spec benchmark suite, information available at <http://www.spec.org>.
- [2] Rajeev Balasubramonian, David H. Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *International Symposium on Microarchitecture*, pages 245–257, 2000.
- [3] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoefflinger, T. Lawrence, J. Lee, D. Padua, Y. Paik, W. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with polaris. *IEEE Computer*, December 1996.
- [4] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *27th Annual International Symposium on Computer Architecture*, pages 83–94, 2000.
- [5] Doug Burger and Todd Austin. The simplescalar tool set, version 2.0.
- [6] Jason F. Cantin and Mark D. Hill. Cache performance for spec cpu2000 benchmarks.
- [7] Calin Cascaval, Luiz DeRose, David A. Padua, and Daniel A. Reed. Compile-time based performance prediction. In *Languages and Compilers for Parallel Computing*, pages 365–379, 1999.
- [8] John H. Edmondson, Paul I. Rubinfeld, Peter J. Bannon, Bradley J. Benschneider, Debra Bernstein, Ruben W. Castelino, Elizabeth M. Cooper, Daniel E. Dever, Dale R. Donchin, Timothy C. Fischer, Anil K. Jain, Shekhar Mehta,

- Jeanne E. Meyer, Ronald P. Preston, Vidya Rajagopalan, Chandrasekhara Somanathan, Scott A. Taylor, and Gilbert M. Wolrich. Internal organization of the alpha 21164, a 300-mhz 64-bit quad-issue cmos risc microprocessor. *Digital Technical Journal*, 7(1), 1995.
- [9] Somnath Ghosh. *Cache Miss Equations: Compiler Analysis Framework for Tuning Memory Behavior*. Princeton University Department of Electrical Engineering, 1999.
- [10] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: An analytical representation of cache misses. In *International Conference on Supercomputing*, pages 317–324, 1997.
- [11] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Architectural Support for Programming Languages and Operating Systems*, pages 228–239, 1998.
- [12] Michael K. Gowan, Larry L. Biro, and Daniel B. Jackson. Power considerations in the design of the alpha 21264 microprocessor. In *Design Automation Conference*, pages 726–731, 1998.
- [13] Koji Inoue, Tohru Ishihara, and Kazuaki Murakami. Way-predicting set-associative cache for high performance and low energy consumption. pages 273–275.
- [14] Aj Kleinosowski John. Adapting the spec 2000 benchmark suite for simulation-based computer architecture research.
- [15] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

- [16] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Pipeline gating: Speculation control for energy reduction. In *ISCA*, pages 132–141, 1998.
- [17] Suleyman Sair Mark. Memory behavior of the spec2000 benchmark suite.
- [18] Allen Kennedy Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. Rice University Department of Computer Science, 1989.
- [19] M.D. Powell, S.H. Yang, B. Falsafi, K. Roy, and T.N. Vijaykumar. Gated-vdd: A circuit technique to reduce leakage in cache memories. *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, 2000.
- [20] Michael D. Powell, Amit Agarwal, T. N. Vijaykumar, Babak Falsafi, and Kaushik Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, 2001.
- [21] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.
- [22] J. Reilly. Spec95 products and benchmarks. *SPEC Newsletter*, 1995.
- [23] Gabriel Rivera and Chau-Wen Tseng. Eliminating conflict misses for high performance architectures. In *International Conference on Supercomputing*, pages 353–360, 1998.
- [24] L. Wei and K. Roy. Design and optimization for low-leakage with multiple threshold cmos. *IEEE Workshop on Power and Timing Modeling*, 1998.
- [25] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer-Ann M. Anderson, Steven W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An

- infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12):31–37, 1994.
- [26] M. E. Wolf and M. S. Lam. A data locality optimization algorithm. In *SIGPLAN '91 Conference on Programming Language Design and Implementation*, 1991.
- [27] Se-Hyun Yang, Michael D. Powell, Babak Falsafi, Kaushik Roy, and T.N. Vijaykumar. An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance i-caches. *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, 2001.
- [28] M. Zhang and K. Asanovic. Highlyassociative caches for low-power processors, 2000.