LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-SR-215372

Statement of Work for Studies in BlueGene/L Scalability and Reconfigurability

A. Henning, S. A. McKee

September 15, 2005

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Subcontract #: B539825 (LLNL/Cornell)
Statement of Work for Studies in BlueGene/L Scalability and Reconfigurability
Milestone 4: Fourth Quarter
September 5, 2005

Amy Henning and Sally A. McKee

As referenced in the subcontract, the work included three major goals: (1) study the performance of an ASCI application, (2) study tradeoffs in using the second CPU in coprocessor mode to optimize use of the L3 scratchpad memory for performing vector-like gather/scatter and streamlining operations, and (3) perform simulator studies of hardware phase detection and identification. We made some modifications to the work contract. Work involving the integration of a cache-conscious data placement algorithm to improve cache utilization on BlueGene/L has been added and work involving the L3 scratchpad memory has been eliminated. This was explained in the previous milestones.

In this milestone, we continue to focus on the last goal by modifying a cycle-accurate simulator, sim-alpha [4]. As premise to hardware phase detection and identification, we need to have an infrastructure for testing various cache-conscious data placement methods. For this milestone, we discuss the completed framework that handles cache-conscious placement optimizations, which includes profiling data accesses and handling remapped addresses. We will also introduce an algorithm (ccdp profiling tool) that we implemented for assigning remapped addresses for a given code. Our performance results show that by using our ccdp profiling tool, we achieve reduced miss rates and an improved overall simulation performance. For our test cases, we use four applications from the SPEC CPU 2000 suite [2].

In our past milestones, we studied research that involves implementing cache-conscious data placement techniques. By becoming more familiar with previous research, we can make better decisions on designing our cache-conscious profiling tool. It is important to have a firm understanding of the existing techniques that have proven to be efficient at improving memory performance, since our tool will produce trace files as input to our enhanced simulator framework.

1 Background

1.1 Previous Approach for Identifying Frequent Item sets

After researching Chilimbi and Hirzel's Dynamic Hot Data Streams [3] technique, which detects and prefetches frequently accessed data streams, we noticed that their algorithm is somewhat limited when it comes to determining access patterns. Their streams are composed of consecutively accessed data. A possible solution is to broaden the range of frequent access patterns by placing fewer restrictions on the access patterns. Instead of defining a stream to consist of only consecutive access, allow for a few interleaved accesses as well. A possible solution is to apply an algorithm like sequential pattern mining, which we discussed in previous milestones. In this section, we give a recap on the idea and problems we encountered when trying to use these pattern mining tools.

The primary goal behind the sequential pattern mining methodology is to solve a problem that involves examining an exponential number of subsequence patterns. It is a challenging problem since it needs to examine a combinatorial, large number of subsequence patterns. Pattern mining techniques can result in extremely long processing time when mining the addresses in cache-access traces for highly memory-bound applications.

There are a number of different algorithms that are developed to optimize this challenging mining process. In general, mining methods adapt a multiple-pass, candidate-generation-and-test approach. Many methods follow the Apriority-like methodology that substantially reduces the working set (i.e., cache-access trace) being examined. The formal definition states that when given a set of sequences, where each sequence consists of a list of elements (i.e., set of addresses), and given a specified minimum support constraint, the mining method will find all of the frequent subsequences that occur no less than the minimum support constraint [7].

In order to allow for faster and more efficient processing, a refined method needs to be utilized. For our initial efforts in finding the most frequent virtual addresses, we chose to explore a sequential pattern mining method called PrefixSpan [7] that focuses on prefix-projection. We initially chose this algorithm since we have a tool, referred to as a frequent itemset tool that is readily available to us and simulates PrefixSpan. In the third milestone, we conducted some experiments using this tool and provided some promising results.

To give a quick overview of this algorithm, PrefixSpan mines the complete list of frequent patterns but also reduces the size of projected databases. This leads to faster and more efficient processing. The goal is to use a more scalable approach to mining. Instead of considering all possible occurrences of frequent subsequences, the projection is based only on frequent prefix subsequences since any frequent subsequence can be found by growing a frequent prefix.

In theory, this approach seems sound. However, having further used the frequent item set tool excessively for identifying frequent addresses in our cache-access traces, we

encountered a few problems. Although the tool follows the algorithm quite accurately, it can only sample a small window of instructions at a time. The tool is not optimized to handle a large database of addresses since it currently uses the entire footprint of a system with a gigabyte of memory.

Even though the frequent items et tool allowed the user to specify constraints like gap width (e.g., maximum number of addresses that can exist between items in a set,), window slider width, and the amount it can slide, it did not identified closed sets which posses another real problem. A single address from a trace would appear in multiple sequences, which gives a false representation of a frequent item set.

Another tool we considered is a fast and memory efficient algorithm, DCI Closed [6], which was recently presented at an IEEE Frequent Itemset Mining Implementations Workshop. This algorithm focuses on our previous problem of duplicate detection of frequent itemset. It first detects and discards these duplicate item sets and displays only closed item sets.

Having tested their tool, it was clear that the algorithm demonstrates efficient memory usage and was proven to be fast at generating results for large traces. We were able to modify the tool to handle 64-bit addresses in order for it to read our alpha memory traces. Unfortunately, we soon discovered that this tool was not going to be a helpful solution. It does not account for addresses that occur within close proximity of each other. Thus, it does not preserve temporal locality. This tool only detects closed item sets for an entire trace, grouping addresses that appear very frequently but do not occur close together in time.

In general, these sequential pattern mining algorithms do not seem to be an efficient resource for detecting frequent addresses for our initial cache placement purposes. We are most interested in preserving temporal locality for addresses in relation to other addresses in a memory trace. The frequent itemset tool that simulated the PrefixSpan approach seemed to have the most potential as a pattern mining tool for our research purposes, but needs to be optimized to be more memory efficient and to detect closed item sets.

1.2 Integrating Temporal Ordering Information

Having temporal locality information on the elements that are accessed around the same time can be a powerful tool. Closely accessed elements can continuously evict each other, causing the system to propagate many cache conflict misses. Being aware of these particular elements can allow for better decisions to be made for data placement.

Temporal relationship graphs are useful in facilitating placement decisions. They are weighted constraint graphs that contain nodes, which can be any element from a virtual address to a procedure call, and edges, which represent the temporal relationship between nodes. Thus, the higher the weighted edge values, the stronger the temporal relationships. This type of temporal information is used in placement algorithms when

nodes are merged together to create a single linear representation of the most frequent elements of the graph.

Gloy and Smith [5] devised a variation of a placement algorithm that uses temporal relationship graphs. For purposes of procedure placement, these temporal relationship graphs are defined for code-block granularity. Their algorithm uses the temporal ordering information that is extracted from a profiler-generated trace, along with instruction cache configuration and procedure sizes, to develop an efficient conflict cost metric. This metric, which was later applied in Calder et al.'s CCDP algorithm [1], is used to quantify an edge value for the graphs. With these edges and nodes set in the graphs, techniques for merging the nodes into one single chain can then be applied. Nodes with heavily weighted edges are used to place procedures in close proximity in the address space.

Having researched this approach, we found it to be quite effective in finding the frequently accessed addresses for our cache placement tool. It is unique to the properties of a cache structure, unlike sequential pattern mining techniques, and can span a large database of elements.

1.3 Approach Selected for CCDP Profiling Tool

In order to validate that our framework for sim-alpha is working properly, it is essential to have a cache-conscious algorithm that will generate remapped addresses for the simulator. Since we do not have access to any tools that automatically conduct cache-conscious data placement techniques, we have to carefully study previous approaches taken and develop our own tool. We looked at a number of previous cache-conscious data placement algorithms and tools that can be used for finding frequent addresses. After much trial-and-error, we decided to take a combination of a couple of approaches.

The cache-conscious data placement algorithm that we developed for testing our framework closely follows the algorithm used by Calder et al. However, there are some subtle modifications. When using the temporal relationship graphs to do data placement, we assign the actual virtual addresses to be the nodes of the graphs. Calder et al. use objects (variables) as nodes to the graphs as well as a customized version of malloc for handling heap addresses. Since our framework for the simulator catalogs all of the virtual addresses that are used by the level-one data cache, we do not have to detect the addresses associated with heap objects. We use the low-level features of the simulator to our advantage.

2 Framework Design and Setup

In this section, we describe the design and implementation of our cache-conscious data placement framework for sim-alpha. This infrastructure is composed of three main components. First, sim-alpha is modified to catalog all data accesses. Second, a CCDP profiling tool is implemented to identify the temporal relationships of these accesses and remap their virtual addresses to minimize cache misses. The data placement algorithm

used in this tool is based on the framework used by Calder et al. Third, sim-alpha is further modified to properly initialize and use the remapped virtual address. The primary goal of this design is to reduce the cache miss rate by remapping memory references in the level-one data cache.

In the next two sections, we will discuss the first and third infrastructure components, which are the actual modifications that are made to the simulator for gathering baseline information that is pertinent to address remapping, and delivering the remapped addresses.

2.1 Collecting Baseline Information

The first step in applying any cache-conscious data placement optimizations is to profile to applications being run in the simulator. Profiling involves gathering all memory accesses to the data cache and cataloging their associated reference types.

In the simulator, we need to identify the pipeline stages where accesses to the level-one data cache occur. There are two stages in Sim-alpha where these accesses occur. In the writeback stage, the dependent chain of completed instructions from the event queue, which holds all instructions that are fetched and issued, are checked and marked "ready" for processing. For our purposes, we are interested in what happens when a load or store is picked from the event queue. From looking at the simulation code and documentation, we discovered that within the writeback stage, load instructions access the data cache and store instructions are marked as "ready" for accessing the data cache. Store instructions will then access the data cache in the commit stage, where instructions are also retired from the reorder buffer and the architectural register file is updated.

When the data cache is accessed from both these stages, a cache function is called. In order to isolate the placement of our code in the simulator, we chose the cache function location. Inside the cache function, we have added code to output the virtual address, used by the data cache, to a file.

Along with extracting the virtual address, we also determine the reference type of this address. We created a separate module, *ccdp*, for sim-alpha that holds all the functions needed for our new framework. There are four reference types that our framework recognizes: stack, constant, global, and heap. The ranges for these reference types can be tracked inside the loader module, which naturally loads the program executable before simulation begins. Static information, such as constants, global and initial stack data, is placed in the virtual memory space at this point. There are important break points that are assigned while the program is being loaded. We use these break points to determine which type of data is associated with the addresses we profile.

After extracting all of the memory accesses of a given application and their type, we can then use that as an input file to our cache-conscious data placement profiling tool.

2.2 Remapping Memory References

The second major feature of our framework is the remapping of memory references. Given a list of virtual addresses and their remapped value, we can selectively remap virtual addresses in the simulator.

Before simulation begins, a hash table needs to be created and initialized. Since there is such a large number of addresses (2^{64}) that exist in the modeled machine, it is important to implement a hash table to handle address lookups. All hash functions that initialize, insert values, and retrieve values from the hash table are located in our ccdp module.

As programs run in the simulator, load and store instructions will be accessing the data cache with their assigned virtual addresses. In order to integrate the remapped addresses, we need to do the virtual address look-up at this same phase in the simulator.

In the cache access function, we perform a lookup in the hash table, which contains all the virtual addresses we translated and their remapped values. Every time the cache access function is called, we check to make sure it is performing a data cache lookup and we check to see if the address exists in the hash table. If it does exist in the hash table, we replace the old address value with the remapped address value. If the address is not in hash table, then we just proceed with the original address.

A caveat of this method is that the remapped memory locations might not be properly initialized. When first attempting to remap addresses in the simulator, one of the remapped addresses would stall the simulator. The simulator would recognize the remapped address as a bogus or misspeculated address. This was occurring since we had not initialized the remapped addresses before running our test programs. Therefore, we created another initialization function in the simulator to handle this problem. When the program's executable is first loaded in the simulator, it copies the data to its virtual memory space and in this process, that memory block is initialized. By calling our initialization function after loading the program and before simulation, we resolved this problem by copying the initial state into the remapped memory locations.

Once the simulator runs a program with our added remapping features enabled, the simulator performs cache-conscious data placement. In the results section of this report, we discuss some of the noticeable performance improves, which include reduced data cache miss rates.

2.3 Cache-Conscious Data Placement Profiling Tool

To determine the most effective addresses for remapping, we need an efficient CCDP tool to profile all the reference to the level-one data cache. These references were gathered during a profile run of our modified simulator. In this section, we will explain the algorithm that was designed to perform cache-conscious data placement and test the remapping functionality of our simulation framework. In this section, we will explain the procedures and functions used for our profiling tool.

Phase 1. Catalog All References:

Before using this tool, a file, which contains all the profiled data cache references and their types, will need to be provided as input to the CCDP tool. This file is provided after the first profile run of the simulator, which we discussed earlier.

The first step in our CCDP algorithm is to create and initialize a hash table that will hold all of the addresses, their reference type, and a frequency counter. After initialization, the input file is scanned and each address is looked up in the hash table. If the address is not found in any of the entries, the address is added to the hash table. If the address does exist in the hash table, then its frequency counter is incremented. The purpose of this hash table is to catalog all references that were used in the simulator's data cache. This table is then used to construct a constraint graph later.

Phase 2. Create Temporal Relationship Graph Structure:

Using the cataloged frequencies of each address in the hash table, a one dimensional array with the top *N* most frequently accessed addresses is created, where the user specifies *N* (default being 5,000). All of these addresses will then be entries in the temporal relationship graph.

The temporal relationship graph is a weighted constraint graph with its nodes being the most frequently accessed references, and its edges being their temporal relationship in the data cache. When first constructing this graph, some data cache properties of the simulator need to be considered.

To begin building the graph, we needed to know the number of addresses that will be remapped and the size of the data cache, which is determined by multiplying the number of sets, blocks, and words of the simulator. This cache size measurement is important when estimating the temporal relationships of the addresses.

The graph structure is represented using a two-dimensional triangular array. Two copies of the graph are created at this early phase. One is used as a reference to the original graph and the other copy is a working graph, which is used to apply changes made after nodes are merged.

Phase 3. Find Temporal Relationships:

After the graphs are constructed, we find the temporal relationships between all the nodes. The method used was also used in Calder et al.'s algorithm and is referred to as the cost conflict metric. The weights of an ordered set of recently referenced addresses are maintained by a queue, which can grow to a maximum size no greater than two times the size of the cache.

Using the given input file, the cost conflict metric processes one address at a time. When an address is processed, the address is first inserted at the front of the queue to preserve ordering of the program memory access patterns. Then the address is walked along the queue until it finds a match or the end of is reached. If a match occurs, it is removed from the queue and all the weights of the edges between this address and addresses between the front of the queue and the match position are incremented. These weights express the degree in which these accesses are interleaved.

Phase 4. Arrange References:

Once the entire trace file has been completely processed, the nodes of the graph need to be arranged in a manner such that addresses with high temporal locality are adjacent to one another. By merging each pair of nodes that share the highest weighted edge, we will eventually be left with a single chain of accesses. A major programming consideration in choosing two-dimensional arrays over more memory-efficient adjacency lists is the complexity of this merging procedure.

When merging nodes, the working graph's edges are searched for the largest weight. The pair of nodes that are associated with this edge are then merged into one single node and the working graph is updated. This update can become a difficult process since we need to keep careful track of the conditions and location of the nodes that surround the pair being merged. When updating the graph, there may be instances where a node has edges that connect to both the two nodes that are currently being merged. For this instance, the weights of the common edges of the merged pair are summed after merging of the pair is completed.

The process for merging nodes in the temporal relationship graph is complete when there is only one node remaining in the graph. This single node is composed of an ordered list of all the nodes in the original graph. This chain of addresses is now ready to be remapped for cache placement.

Phase 5. Determining Cache Placement:

Using the user-specified configurations of the data cache, the final chain of addresses that remain after merging is then broken up into contiguous chunks of addresses that are the size of a cache line. This process apply to either direct-mapped or set associative caches.

Since we remap addresses by grouping reference types, four distinct sections of the simulator’s virtual memory space are designated based on their type. Since the simulator uses the bottom portion of memory, the top half (i.e., 0x1000000000000000 and up) is solely used for remapped addresses that we assign with this tool. The top three bits of the remapped address, will represent the type of reference being remapped, thus pointing to four large partitions of the top half of the virtual memory.

2.4 Validating the Framework

To validate that our framework is functioning properly and producing correct results, we ran a few microbenchmarks [4] that are provided with the Sim-alpha package. These benchmarks test three architectural features of the simulator: the control flow, execution core, and memory system. The level-one and level-two cache latency and the main memory latency are evaluated in the memory system tests.

We first ran the microbenchmarks on the baseline simulator and collected all statistical information including the output for the programs. With these statistics in hand, we then can make direct comparisons to our framework version. Since there are two main features to our framework, (1) collecting the original virtual addresses and (2) remapping the addresses, we ran the microbenchmarks twice.

We checked such parameters as the total number of instructions that were executed and committed and the total number of loads and stores issued. The statistical information matched for both runs, and we also generated the same output for all benchmarks that did not report timing measurements. The runs that gathered the original addresses had the same latency measurements as the baseline simulator. For the runs that performed the remapping, slightly reduced elapsed time and latency were displayed as well as an increase in memory bandwidth, which is to be expected with applications that have fewer cache misses.

3 Results

The default configurations in alpha.cfg and mem.cfg were used in our simulations. They are supplied with the sim-alpha package. One major alteration was the data cache. We modified it to be a direct-mapped cache with 1024 sets at 64 bytes per set. This is also similar to the level-one data cache for BlueGene/L. An overview of the memory configurations is listed in the table below.

L1 instruction cache	64 KByte 2-way SA
L1 data cache	64 KByte direct-mapped
Victim buffer	8 entries
L2 unified cache	2 MByte direct-mapped
L1 instruction cache latency	1 cycle
L1 data cache latency	3 cycles
L2 latency	7 cycles

Table 1: Sim-alpha Memory System Parameters

To simplify the workload, we used Simpoint to find phases of the codes that best represent the execution behavior of the entire test programs. For our test cases, we generated simpoints with a k-max of 30 and interval size of 10 million executed instructions.

In order to run these individual simpoints, the simulator needs to be run using the fast forwarding feature. This can be specified on the command line along with the interval of instructions to skip and the simpoint where simulation will start. When running the simulator with fast forwarding enabled, the simulator will skip all of the instructions up to the simpoint. When it reaches the simpoint, the simulator starts executing the code until the entire simpoint has been executed, which in our case is 10 million instructions. All of the SPEC CPU benchmarks were profiled and remapped using simpoints with fast forwarding enabled. We evaluate the entire benchmark by combining the statistics for each of their simpoints and distributing them with their weight, which is given by the Simpoint tool.

In this section, we present performance measurements for a direct-mapped data cache. The graphs show our experimental results for the SPEC CPU 2000 *gcc*, *equake*, *art*, and *ammp*. The first figure demonstrates the differences in miss rate between the baseline and the remapped version of the simulator. The second figure shows the performance of the simulator in elapsed time.

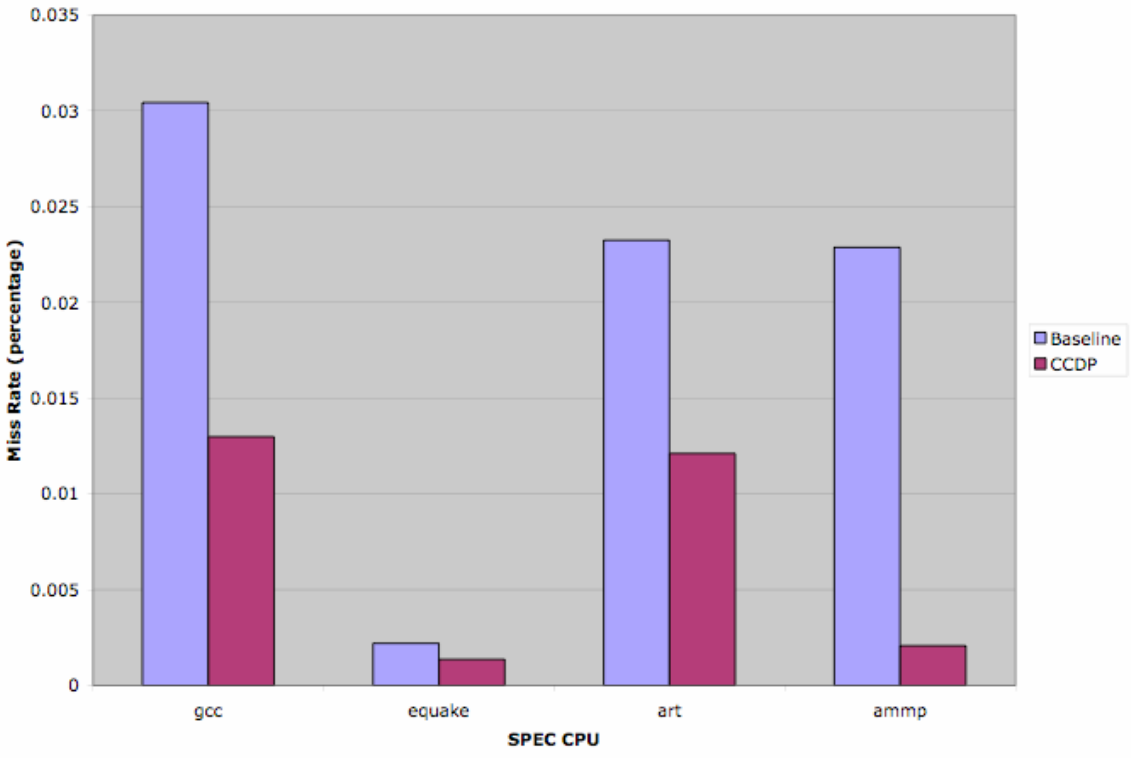

Figure 1: Sim-alpha L1 Data Cache Miss Rate

The results from both experiments show that by using a cache-conscious data placement layout of the accesses to the level-one data cache, we can obtain better performance. It is clear that when using a direct-mapped cache configuration, the system has a lower miss rate when using cache-conscious data placement. There is a 9-90% decrease in miss rate for these four test cases. Since each benchmark has its own distinct input data set and has different execution behavior, our remapping tool will affect each benchmark differently. Some benchmarks have more distinct access patterns that occur frequently, thus allowing for better cache utilization when applying the remapped addresses. For the overall performance of the simulator, there was a 14-33% improvement.

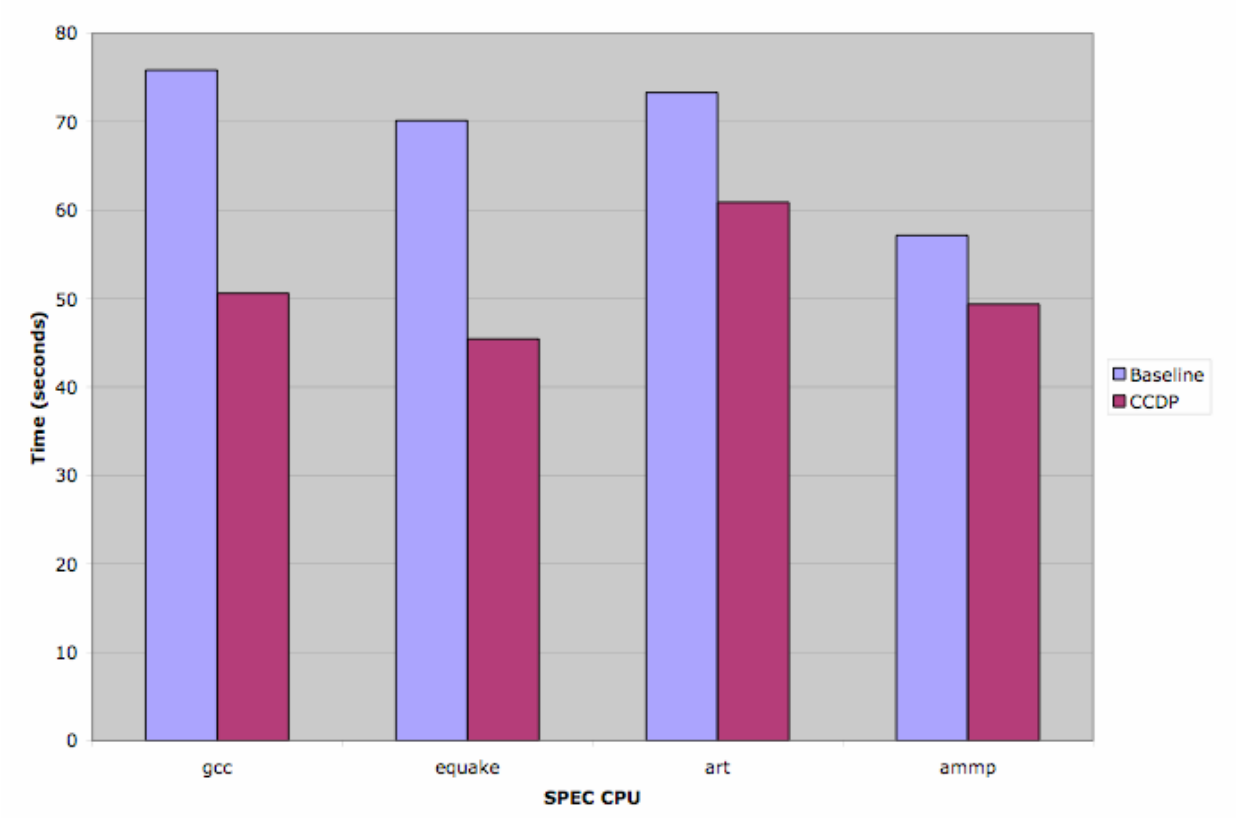

Figure 2: Sim-alpha Runtime Performance

4 Conclusions

This design project involved modifying a cycle-accurate simulator to perform cache-conscious data placement optimizations and constructing a scheme for verifying its correctness. Using such an optimization can potentially improve the cache performance by reducing the number of cache misses.

In this design project, we were able to complete this framework. There are three major components for the overall project. First, the sim-alpha simulator was modified to catalog all data accesses. Second, a profiling tool was fully implemented to identify the temporal relationships of these accesses and remap their virtual addresses to minimize cache misses. Third, sim-alpha was further modified to properly initialize and use the remapped virtual address.

To verify that the framework was running properly, we used the same microbenchmark suite that was used for verifying the correctness for sim-alpha simulator. Using this benchmark suite, we successfully validated that the program output and simulations are correct.

For our performance results, we used four applications from the SPEC CPU 2000 benchmark suite. We gathered statistics relevant for the level-one data cache miss rates as well as the simulation elapsed time. For all statistics gathered, there was an improved for all four of the benchmarks. For the miss rates, there was a decrease ranging from 9-90% and for the simulation elapsed time, there was an overall speedup from 14-33%. From our experiments, we show that using a cache-conscious data placement optimization can improve system performance.

With this framework available, we can begin using the simulator to apply our studies of phase detection. Other cache-conscious data placement algorithms can also be tested using this framework.

Bibliography

[1] B. Calder, C. Krintz, S. John, and T. Austin. Cache-Conscious Data Placement. In *Proceedings of the 8th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 139–149, October 1998.

[2] J. F. Cantin and M. D. Hill. Cache Performance for SPEC CPU2000 Benchmarks. `http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data`, May 2003.

[3] T. Chilimbi and M. Hirzel. Dynamic Hot Data Stream Prefetching for General-Purpose Programs. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–209, June 2002.

[4] R. Desikan, D. Burger, S. Keckler, and T. Austin. Sim-alpha: A Validated, Execution-Driven Alpha 21264 Simulator. Technical Report TR-01-23, Department of Computer Sciences, The University of Texas at Austin, 2001.

[5] N. Gloy and M.D. Smith. Procedure Placement Using Temporal-Ordering Information. ACM Transactions on Programming Languages and Systems, 21(5):977–1027, 1999.

[6] C. Lucchese, S. Orlando, and R. Perego. DCI Closed: A Fast and Memory Efficient Algorithm to Mine Frequent Closed Itemsets. In *Proceedings of the 2004 IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, November 2004.

[7] J. Pei, J. Han, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth. In *Proceedings of the 17th IEEE International Conference on Data Engineering*, pages 215–226, April 2001.