HiDISC: A Decoupled Architecture for Applications in Data Intensive Computing

Drs. Alvin Despain and Jean-Luc Gaudiot

Final Report

Abstract

The ever growing speed gap between processor and main memory has been a major performance bottleneck of modern computer systems. As a result, today's data intensive applications suffer from frequent cache misses and lose many CPU cycles due to pipeline stalling. Although traditional prefetching methods reduce cache misses considerably, most of them strongly depend on the access pattern being predicted and fail when faced with irregular memory access patterns with low locality.

This report presents our design and performance evaluation of a novel, high-performance decoupled architecture called HiDISC (Hierarchical Decoupled Instruction Stream Computer). HiDISC provides low memory access latency by introducing enhanced data prefetching techniques at both hardware and software levels. Three dedicated processors for each level of the memory hierarchy act in concert to mask the memory latency.

As required by the DARPA Data Intensive program, we used as our performance evaluation benchmarks the Data-intensive Systems Benchmark Suite and the DIS Stressmark suite. The simulation results for both benchmarks show a distinct advantage of the HiDISC system over current prevailing superscalar architectures.

Table of Contents

List of Figures

List of Tables

1. Introduction

The speed mismatch between processor and main memory has been a major performance bottleneck in modern processor architectures. Processor speed has been improving at a rate of 60% per year during the last decade. Conversely, access latency to main memory has been improving at less than 10% per year [24]. This speed mismatch – the Memory Wall problem - results in considerable cost in terms of cache misses and severely degrades processor performance. The problem becomes even more acute when faced with highly data intensive applications. Indeed, these applications are becoming more prevalent. By definition, they have a higher memory access/computation ratio than "conventional" applications. Moreover, the access pattern tends to be more irregular. As a result, the penalty caused by cache misses is becoming even more serious. This means that the architect must either reduce pipeline stalling upon cache misses or reduce the number of those cache misses (incidentally, this latter objective is the main goal of the HiDISC project).

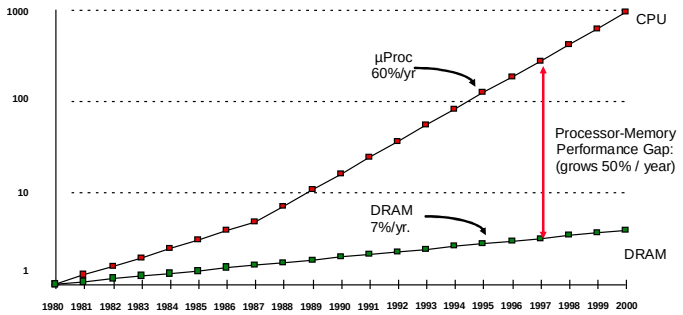

Figure 1: The speed mismatch between CPU cycle and DRAM speed

Reaching higher Instruction-Level Parallelism (ILP) through multiple instruction issue and out-of-order execution has been an essential part of modern processor design for many years. Moreover, sophisticated branch prediction and speculative execution techniques provide more opportunities for the discovery of independent instructions across basic blocks [31]. Various approaches using Thread-Level Parallelism (TLP) have also been introduced to deliver more ILP. During the last decade, superscalar and very

long instruction word (VLIW) architectures have played an important role in ILP research. Although both models are designed to deliver higher levels of parallelism through multiple instruction issue, the ever increasing memory access latency has become a major obstacle to the exploitation of higher degrees of ILP.

To solve the *memory wall* problem, current high performance processors are designed with large amounts of integrated on-chip cache. However, this large cache strategy works efficiently only for applications which exhibit sufficient temporal or spatial locality. Newer applications such as multi-media processing, database, embedded processor, automatic target recognition, and any other data intensive programs exhibit irregular memory access patterns [15] and result in considerable numbers of cache misses which cause significant performance degradation.

To reduce the occurrence of cache misses, various prefetching methods have been developed. Prefetching is a mechanism by which data is fetched from memory to cache before it is even requested by the CPU. It can be implemented either in hardware or in software. Hardware prefetching [6] dynamically adapts to the runtime memory access behavior and decides the next cache block to prefetch. Software prefetching [20] usually inserts the prefetching instructions inside the code. Although previous prefetching research considerably contributed to improvements in cache performance, prefetching techniques still suffer from irregular memory access patterns. Indeed, typical prefetching strategies strongly depend on the predictability of the future data addresses. This is very difficult to predict when the access patterns are random [19]. Moreover, many current applications use sophisticated data structures with pointers which dramatically lower the regularity of memory accesses.

The Data Intensive Systems Benchmark Suite and the DIS Stressmark Suite are used in this project as our performance evaluation benchmarks. Both benchmarks are provided by Atlantic Aerospace Electronics Corporation [38][39] and supported by the Data Intensive Systems project of the DARPA Information Technology Office. Stressmark includes seven small data intensive benchmarks. Conversely, the DIS

benchmarks consist of five codes more realistic than Stressmark. The five benchmarks can be categorized into three groups:

1. The Model based image generation group has two benchmarks – Method of Moments and Simulated SAR Ray Tracing.

2. The Target detection includes Image Understanding and Multidimensional Fourier Transform.

3. The Data Management benchmark

2. Method, Assumptions, and Procedures

In order to counter the inherently low locality in Data Intensive applications, our design philosophy is to emphasize the importance of memory-related circuitry and even employ two dedicated processors to respectively manage the memory hierarchy and prefect the data stream.

2.1 The HiDISC System

Access/Execute decoupled architectures have been developed as alternate processor architectures which exploit the parallelism between data access operations and "normal" computation. Concurrency is achieved by separating the original, single instruction stream into two streams based on the functionality of instructions. Asynchronous operation of the streams provides for a certain distance between the streams and makes data prefetching possible. The HiDISC architecture is an enhanced variation of conventional decoupled architectures.

Decoupled architectures (also called Access/Execute architectures) deliver higher degrees of Instruction-Level Parallelism by separating the sequential code into two instruction streams - *Access Stream* and *Execute Stream* - based on memory access functionality. Each stream runs almost independently of the other. The model was originally developed to tolerate long memory latencies: hopefully, the Access Stream will run ahead of the Execute Stream in an asynchronous manner, thereby allowing timely

prefetching. It should be noted at this point that an extremely important parameter will be the "distance" between the instruction currently producing a data element in the Access Stream and the instruction which uses it in the Execute Stream. This is also called the *slip distance*, and it will be shown how it is a measure of tolerance to high memory latencies. Communication is achieved via a set of FIFO queues (they are architectural queues between the two processors to guarantee the correctness of program flow).

Our HiDISC (Hierarchical Decoupled Instruction Stream Computer) architecture is a variation of the traditional decoupled architecture model. In addition to the two processors of the original design, the HiDISC comprises one more processor for data prefetching [6][8] (Figure 2). A dedicated processor for each level of the memory hierarchy timely supplies the necessary data for the above processor. Thus, three individual processors are combined in this high-performance decoupled architecture. They are used respectively for computing, memory access, and cache management:

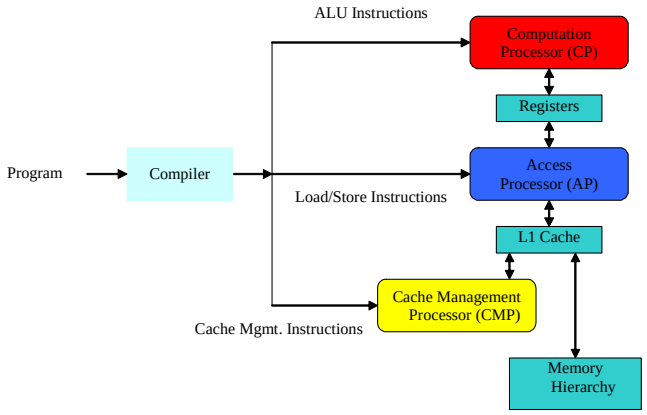

Figure 2: The HiDISC System

- Computation Processor (CP): executes all primary computations except for memory access instructions.

- Access Processor (AP): performs basic memory access operations such as loads and stores. It is responsible for passing data from the cache to the CP.

- Cache Management Processor (CMP): keeps the cache supplied with data which will be soon used by the AP and reduces the cache misses, which would otherwise severely degrade the data preloading capability of the AP.

By allocating additional processors to each level of the memory hierarchy, the overhead of generating addresses, accessing memory, and prefetching is removed from the task of the CP: the processors are decoupled and work relatively independently of one another.

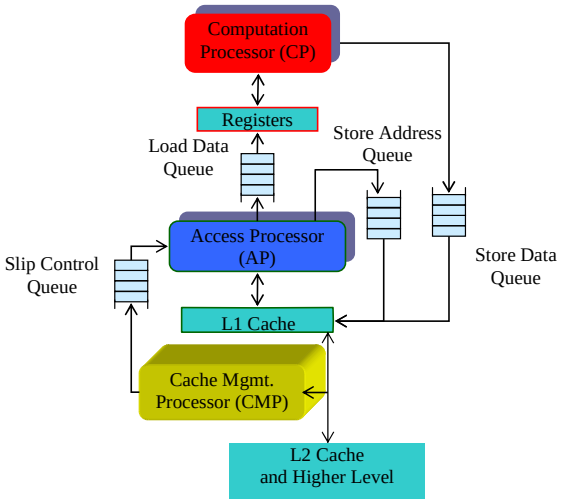

Figure 3: Inside the HiDISC architecture

Now, our compiler must appropriately form three streams from the original program: the computing stream, the memory access stream, and the cache management stream are created by the HiDISC compiler and stored into the program memory of each of the processors. As an example, Figure 4 shows the stream separation for the inner loop of the discrete convolution algorithm.

The control flow instructions are executed by the AP. Incidentally, it should be noted that additional instructions are required in order to facilitate the synchronization between the processors. Also, the AP and the CP use specially designed tokens to ensure correct control flow: for instance, when the AP terminates a loop operation, it simply

deposits the End-Of-Data (EOD) token into the load data queue. When the CP sees an EOD token in the load data queue, it exits the loop.

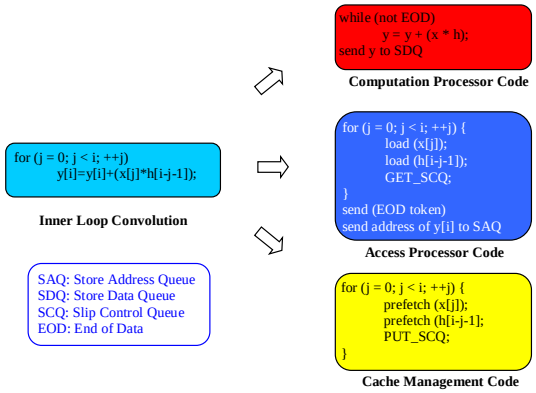

Figure 4: Discrete Convolution as processed by the HiDISC Compiler

2.2 Experimental Environment

In order to evaluate the performance of our proposed architecture, we have designed a simulator for our HiDISC architecture. It is based on the SimpleScalar 3.0 tool set [5] and it is an execution-based simulator which describes the architecture at a level as low as the pipeline states in order to accurately calculate the various architectural delays.

Figure 5 shows a high-level block diagram of the simulation procedure. Each benchmark program follows the two steps described. The first step consists in compiling the target benchmark using the HiDISC compiler which we have designed, while the second step is the simulation and performance evaluation phase.

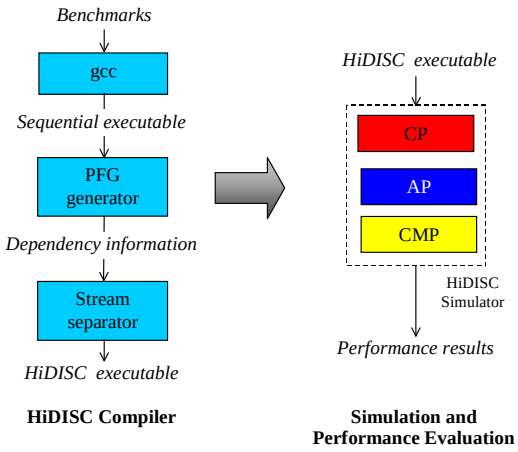

Figure 5: Simulation Procedure

2.3 Operation of the HiDISC Compiler

The HiDISC executables are produced by our HiDISC compiler. The core operation of the HiDISC compiler is stream separation. Stream separation is achieved by backward chasing of load/store instructions based on the register dependencies. This means that, in order to obtain the register dependencies between instructions, a Program Flow Graph (PFG) must be derived. Indeed, the PFG generator and the stream separator are two major operations of the HiDISC compiler. The PFG generator and the stream separator are adopted after some modifications from the SimpleScalar 3.0 tool set and integrated in the HiDISC compiler.

Figure 6 depicts the overall HiDISC compiler. Its detailed operation is described below.

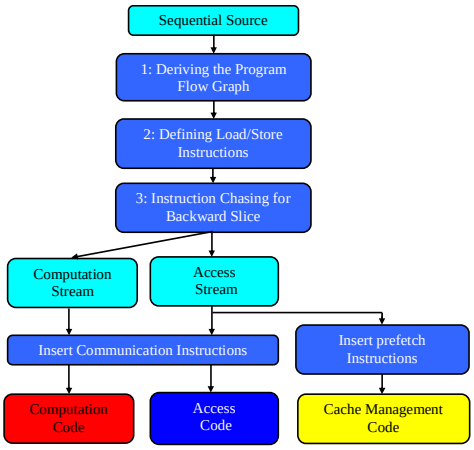

Figure 6: Overall HiDISC stream separator

The input to the HiDISC compiler is a conventional sequential binary code. The first step (1: *Deriving the Program Flow Graph* in Figure 6) consists in uncovering the data dependencies between the instructions. Each instruction is analyzed so as to determine which its parent instructions are. This determination is based on the source register names. Whenever the stream separator meets any load/store instruction in step 2 (2: *Defining Load/Store Instructions*), it defines the instruction as the Access Stream (AS) and chases backward to discover its parents instruction. The next step (3: *Instruction Chasing for Backward Slice*) is designed to handle the backward chasing of pointers. The instructions which are chased according to the data dependencies are called the *backward slice* of the instruction from which we started.

Since the Access Stream should contain all access-related instructions, as well as the address calculation and index generation instructions, the backward slice should be included in the Access Stream as well. It should be noted that all the control-related instructions are also part of the Access Stream. The instructions which should belong to the control flow are determined by a similar method. After defining all the Access Stream, the remaining instructions are, by default, classified as belonging to the Computation Stream (CS).

In addition to the stream separation, appropriate communication instructions should be placed in each stream in order to synchronize the two streams. Finding what

the required communications are is also based on the register dependencies between the streams. Essentially, when it is determined that some required source data is produced by the other stream, some kind of communication should take place. For instance, when a memory load (inside the Access Stream) produces a result which should be used by the Computation Stream, a Load instruction would be inserted in the Access Stream. It would send the data to the Load Data Queue (LDQ). However, if the result of that load was not needed by the Computation Stream, then obviously no such insertion would be needed. Similarly, when the result produced by the Computation Stream is used by a store instruction (inside the Access Stream), it should be sent to the store data queue (SDQ) by inserting an appropriate communication instruction.

The backward chasing starts whenever we encounter new load/store instruction. The backward chasing ends when the procedure meets any instruction which already has been defined as the Access Stream. The parent instructions of any defined Access Stream have already been chased.

After separating the Access Stream and the Computation Stream, the CMP stream is constructed by modifying the Access Stream. The instruction stream for the CMP is indeed quite similar to the Access Stream. Only the load instructions are replaced with the prefetch instructions for the CMP stream.

Figure 7 shows an example of the operation of the backward slicing mechanism in the HiDISC compiler. The assembly code input to the HiDISC compiler is the PISA (Portable Instruction Set Architecture) which is the instruction set of the SimpleScalar simulator [5]. We have selected for this example the inner product of Livermore loop (lll1). The PISA code is compiled into SimpleScalar binary by first using a version of *gcc* which targets SimpleScalar.

Initially, each memory access instruction is defined as belonging to the Access Stream. For example, the ***l.d*** instruction in the fifth line (pointed to by an arrow ① in the left margin) can be immediately determined as belonging to the AS. Moreover, every parent instruction of a memory access instruction should be identified. In the example, the ***addu*** instruction in the fourth line (pointed to by an arrow ②) - due to the register $9

- and the ***mul*** instruction in the second line (pointed to by an arrow ③) -due to the register $25 - are also chased and marked as belonging to the AS. Likewise, other instructions are examined based on the above approach. The instructions in the shaded box in Figure 7 belong to the Access Stream.

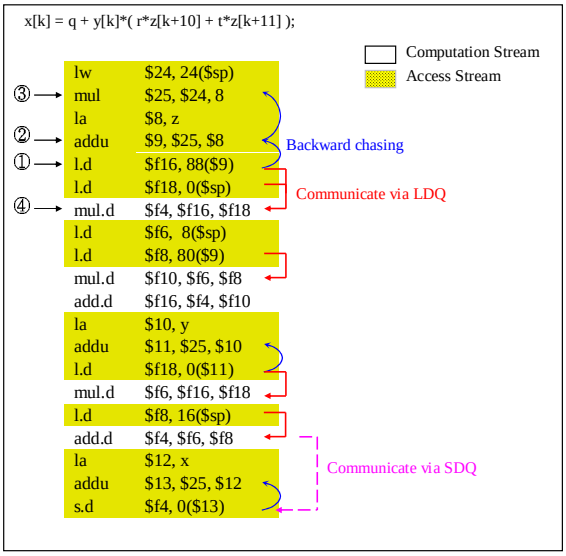

Figure 7: Backward chasing of load/store instructions

After defining each stream, the communication instructions should be inserted. The red lines in Figure 7 (forward arrows, solid lines) show the necessary communications from the AS to CS. For example, the ***mul.d*** instruction (which is marked as being inside the Computation Stream, pointed to by arrow ④) in the seventh line requires data from the other instruction stream (The Access Stream). Therefore, both ***l.d*** instructions in the fifth and sixth line need to send data to LDQ. Likewise, the purple line at the bottom (forward arrow, dotted line) also shows the communication from the CS to the AS via the Store Data Queue (SDQ).

Figure 8 shows the complete separation of the two streams and insertion of the communication instructions.

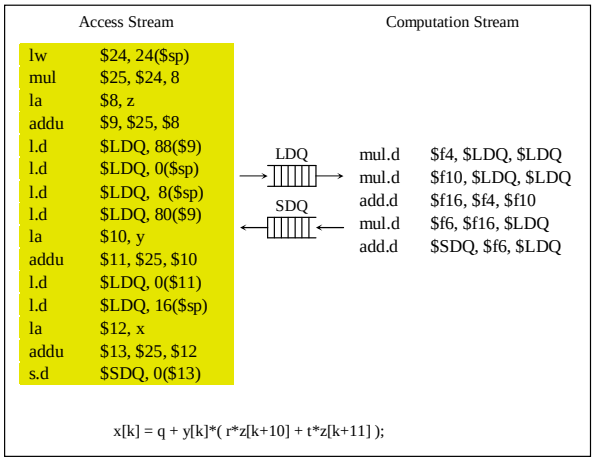

Figure 8: Separation of sequential code

2.4 Benchmark Description

Applications causing large amounts of data traffic are often referred to as data-intensive applications as opposed to computation intensive applications. Inherently, data-intensive applications use the majority of the resources (time and hardware) to transport data between the CPU and the main memory. The tendency for a higher number of applications to become data intensive has become quite pronounced in a variety of environments [39]. Indeed, many applications such as Automatic Target Recognition (ATR) and database management show non-contiguous memory access patterns and currently result in idle processors due to data starvation. These applications are more stream-based and result in more cache misses due to lack of locality.

Frequent use of memory dereferencing and pointer chasing also creates an enhanced pressure on the memory system. Pointer-based linked data structures such as lists and trees are used in many current applications. For one thing, the increasing popularity of Object Orient Programming correspondingly increases the underlying use of pointers. Due to the serial natural of pointer processing, memory accesses become a severe performance bottleneck of existing computer systems. Flexible, dynamic construction allows linked structures to grow large and difficult to cache. At the same

time, linked data structures are traversed in a way that prevents individual accesses from being overlapped since they are strictly dependent upon one another [26].

The applications for which our HiDISC is designed are obviously data intensive programs, the performance of which is strongly affected by the memory latency. As required by the Data Intensive Systems project of the DARPA Information Technology Office, we used for our benchmarks the Data-intensive Systems Benchmark Suite [39] and DIS Stressmark Suite [38] provided by the Atlantic Aerospace Electronics Corporation. Both of the benchmarks are targeting data intensive applications. The DIS benchmarks are five benchmarks codes, which are more realistic and larger than Stressmark. Stressmark includes seven small data intensive benchmarks, which extracts and shows the kernel operation of data intensive programs.

Due to problems with the input data file, the Image Understanding benchmark cannot be executed. Also, since the Corner-Turn benchmark among seven Stressmarks is not provided with the source code, we only simulated the other six Stressmarks.

Table 1 shows the characteristics of each of the benchmarks simulated.

Table 1: Simulated Benchmark Description

Benchmark	Name	Problem	Characteristic
DIS benchmarks	Method of Moments	Computing the electromagnetic scattering from complex objects	Containing computational complexity and requesting high memory speed
	Multidimensional Fourier Transform	Fourier Transform	Wide range of application usage
	Data Management	Traditional DBMS processing	Index algorithms and ad hoc query processing
	SAR Ray Tracing	SAR image simulation	Utilizes Image-domain approach

Stressmark	Pointer	Pointer following	Small blocks at unpredictable locations. Can be parallelized
	Update	Pointer following with memory update	Small blocks at unpredictable location
	Matrix	Conjugate gradient simultaneous equation solver	Dependent on matrix representation Likely to be irregular or mixed, with mixed levels of reuse
	Neighborhood	Calculate image texture measures by finding sum and difference histograms	Regular access to pairs of words at arbitrary distances
	Field	Collect statistics on large field of words	Regular, with little re-use
	Transitive Closure	Find all-pairs-shortest-path solution for a directed graph	Dependent on matrix representation, but requires reads and writes to different matrices concurrently

3. Results and Discussion

We used our architectural simulator of the HiDISC machine to evaluate the performance of all the benchmarks except two.

3.1 Simulation Parameters

In our benchmark simulations, we assumed the architectural parameters outlined in Table 2. The baseline architecture for the comparison is a 4-way superscalar architecture, which is implemented as *sim-outorder* in the SimpleScalar 3.0 tool set. In both cases, the memory access latency has been made to vary between 20 and 120 CPU cycles. The baseline superscalar architecture supports out-of-order issue with 16 register update units and 8 load store queues.

Table 2: Simulation Parameters

Branch predict mode	Bimodal
Branch table size	2048
Issue width	4
Window size for superscalar	RUU: 16 LSQ: 8
Slip distance for AP/CP	50
Data L1 cache configuration	128 sets, 32 block, 4 -way set associative , LRU
Data L1 cache latency	1
Unified L2 cache configuration	1024 sets, 64 block, 4 - way set associative, LRU
Unified L2 cache latency	6
Integer functional unit	ALU(x 4), MUL/DIV
Floating point functional unit	ALU(x 4), MUL/DIV
Number of memory port	2

3.2 Benchmarks Results

Figure 9 and Figure 10 show the simulation results of the DIS Benchmark Suite and the Stressmark Suite. The performance results of the HiDISC architecture are compared to a 4-way superscalar architecture. The far left bar indicates the performance results of the superscalar architectures. The second bar expresses the performance results of the basic HiDISC architecture. The remaining two bars show the possible performance results when enhancing the prefetching capability of the CMP processor. The numbers in parenthesis express the cache miss reduction ratio. The enhancements will be explained in more detail in the next section.

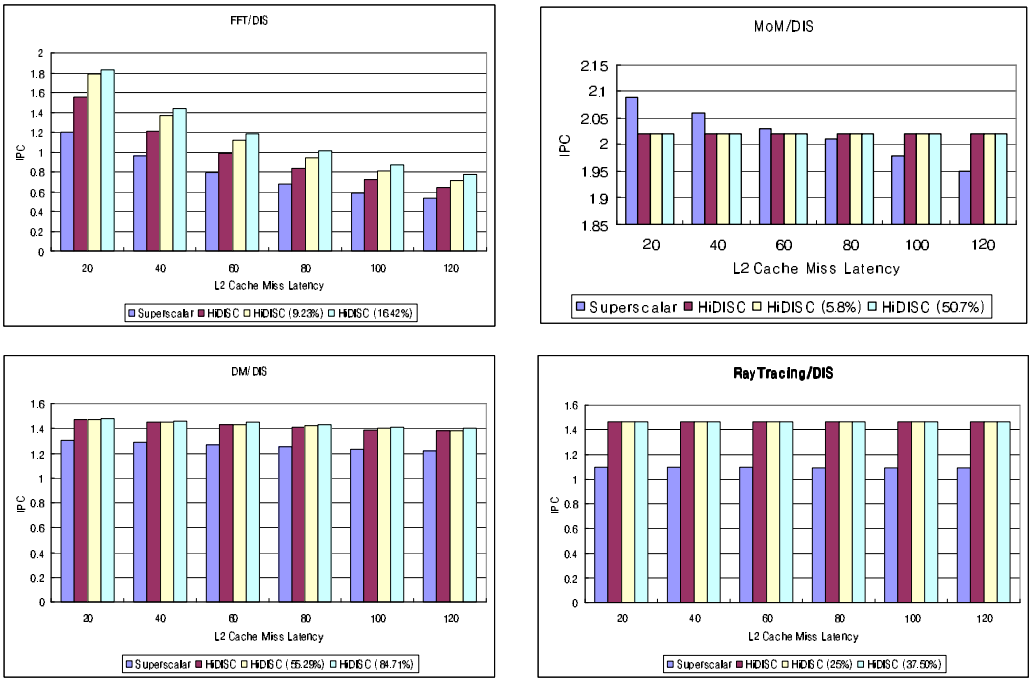

Figure 9: DIS benchmark performance results

All four DIS benchmarks show better performance than the baseline superscalar architecture. However, with the Stressmark, only two of the six cases show better performance for the HiDISC. The remaining four benchmarks do not show any performance advantage for the HiDISC architecture.

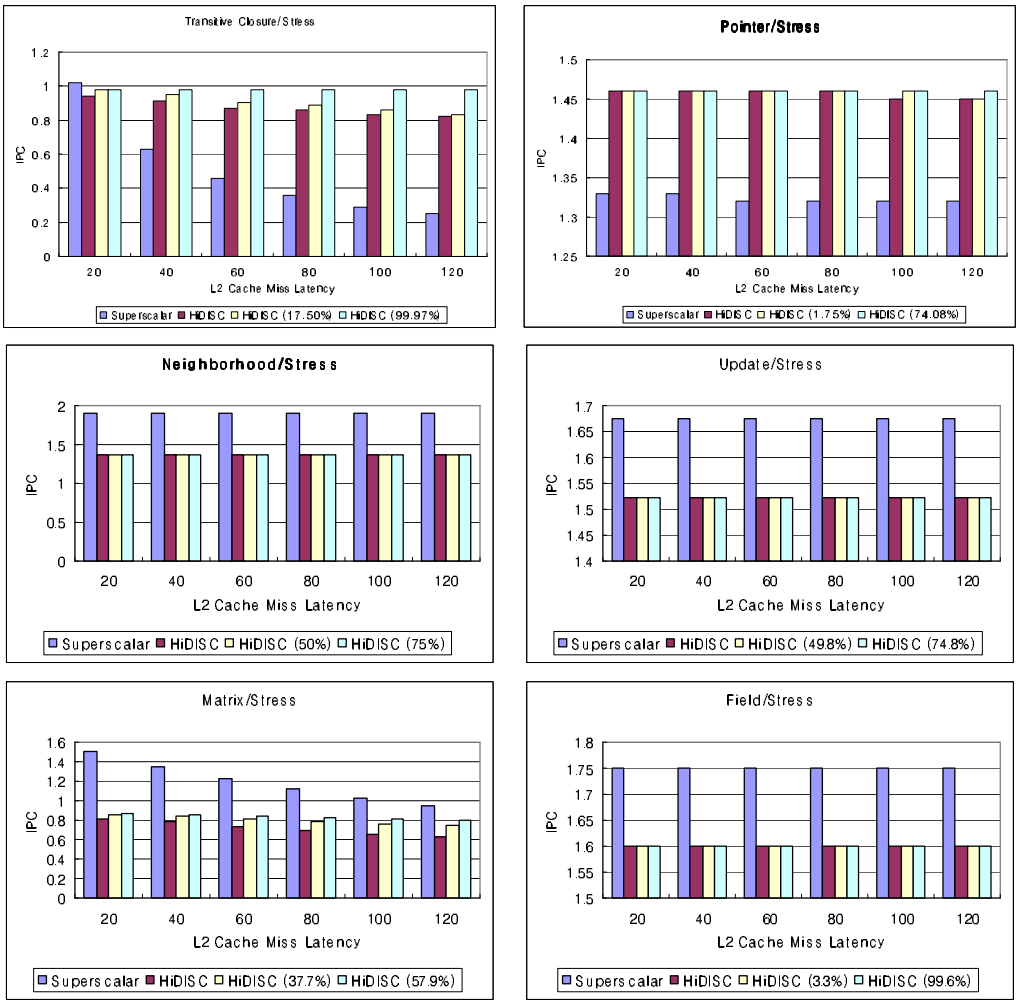

Figure 10: Stressmark performance results

3.3 Discussion

The simulation results show that the HiDISC system performs quite well in general with the DIS benchmarks. This is because the DIS benchmarks contain many long latency floating-point operations which can effectively hide any long memory latency. In other words, the amount of computation code and that of memory access code are well balanced in the DIS benchmark Suite. Conversely, the size of the Stressmark computation code is much smaller than that of the memory access code. It is one of the

main reasons for the somewhat weaker performance results observed in the case of the Stressmark Suite.

Four DIS Benchmarks Results (Figure 9)

Four DIS benchmarks outperform the baseline superscalar architecture particularly with higher memory latencies. More particularly, the Method of Moments is quite robust when faced with longer memory latencies. It contains enough computation code which can hide the longer access latency. Also, the dependencies between the Computation Stream and the Access Stream are comparatively not heavy and provide enough slip distance to hide any long memory latency.

In the case of the Multidimensional Fast Fourier Transform, HiDISC also outperforms the superscalar architecture. However, the results show a weaker performance for long memory latencies even with the HiDISC model. Indeed, the synchronization between the AS and the CS limits the possible slip distance between the two streams. It is due to the data dependencies between the two streams: frequent data dependencies between the Access Stream and the Computation Stream cause *loss of decoupling* events. Usually, it is the CS which has to wait for a data element to be produced by the AS (although the converse is also sometimes true). When this happens, the slip distance between the two processors is reduced significantly, one processor must wait for the other and any advantage is negated since there is no more parallelism between the two processors.

The Data Management and the Ray-Tracing benchmarks are not affected by longer memory latencies in either case. It should be noted that the working set for the Data Management benchmark fits quite well in the cache. As should be expected, a program with a small working set is not a good candidate for a prefetching architecture such as the HiDISC. Conversely, due to the prefetching of the CMP, FFT exhibits better performance.

Six Stressmarks Results (Figure 10)

Generally, the Stressmark codes are too small and contain too many operations which are concerned only with data access. Therefore, the amount of computation code to hide data access is not sufficient. The HiDISC produces weaker results in four Stressmarks – Update, Field, Matrix and Neighborhood - out of the six Stressmarks. However, the remaining two Stressmarks - the Pointer and the Transitive Closure – advantageously exploit the characteristics of our architecture.

Besides the unbalanced computation and access code ratio, frequent *loss of decoupling* is another main reason for the weak performance we observe in several Stressmarks. Indeed, four Stressmarks except Pointer and Transitive Closure contain too much data dependencies and frequent synchronizations between two streams.

However, in the Pointer Stressmark case, pointer chasing can be executed far ahead since it does not require the computation results from the CP. The Transitive Closure benchmark also produces good results because not much in the AP depends on the results of the CP. In both cases, the Access Stream can run far ahead of the Computation Stream: a sufficient slip distance is guaranteed in both benchmarks.

The slip distance is truly inherent to the instruction mix pattern of the application: if the Access Stream does not depend much on results from the Computation Stream, the Access Stream can run earlier and maintain a high slip distance. Pointer and Transitive Closure exhibit good performance for the same reasons. In addition to the possible slip-distance between the two streams, the Stressmark results suggest that applications which are ideal for the HiDISC would be well balanced in terms of the ratio of computation operations over memory operations.

Finally, the working set for the Stressmark is quite small and the baseline superscalar architecture does not suffer from many cache misses. Three Stressmarks (Update, Field and Neighborhood) cannot improve even with the prefeching of the CMP.

Although some of the benchmarks show weak performance, the fact that the Pointer Stressmark and the Transitive Closure Stressmark perform better that the baseline

superscalar architecture is quite encouraging and suggests the type of the candidate applications for the HiDISC architecture.

4. Conclusions

Current high-level programming languages and all supporting compilers are based on an underlying sequential programming behavior. This is confirmed at the lower level where the instruction set of modern microprocessors are based on a sequential model. However, in order to exploit some parallelism at the instruction level, manufacturers of current prevailing high performance processors have considerable changed the processor internal structure. Also, several features of dataflow models have found their way in modern processor architectures and compiler technologies such as register renaming and dynamic scheduling [17]. Decoupled architecture is one such technique which promises to bring improvement to the performance.

The effectiveness of the HiDISC decoupled architecture has been demonstrated here with data intensive applications. It has been eloquently shown that the proposed prefetching method provides better ILP compared to conventional superscalar architectures. However, the possible *loss of decoupling*, which is inherited from the sequential behavior of the programs, stalls the processors and drops utilization in some cases. The results also point to some future modifications of the current CMP for effective prefetching.

Clearly, the HiDISC architecture, as designed, will shine when executing data intensive applications because they contain enough computation to hide long memory latencies. In addition to that, the slip distance is another important factor which determines overall performance. Too many data dependencies of the access processor on the computation processor prevent a sufficient slip distance from developing. Therefore, stream-like applications are favored for the HiDISC system.

5. Recommendations

Based upon these performance results, we propose some improvements to the basic HiDISC architectures in order to make it fit a wider variety of applications.

5.1 Future Enhancements to the HiDISC

Although the independent management of the memory hierarchy provides an opportunity to implement novel prefetching techniques, the HiDISC architecture suffers from two significant weaknesses. First, the frequent synchronizations between the AP and the CP cause stalling of the processors and result in low utilization. Second, the CMP code is essentially not different from the AP code. Therefore, all the load instructions are forced to run on the CMP as prefetching. However, not every prefetching by the CMP is necessary and helpful. Necessary enhancements regarding the above two problems will follow.

The frequent synchronizations cause *loss of decupling* and prevent timely prefetching. Therefore, each processor of the HiDISC loses many CPU cycles to wait until the necessary data arrives. To solve this problem, Simultaneous MultiThreading (*SMT*) should be added to the HiDISC architecture. SMT will raise the utilization by running multiple threads simultaneously. In other words, in a multithreaded HiDISC system, SMT would raise the utilization of the processors, while decoupling would reduce the memory latency [22][23].

The second modification is related to the current CMP design. The main motivation for the existence of the CMP processor is to reduce the cache miss rate by the Access Processor by timely prefetching. Therefore, the CMP should run ahead of the AP, just like the AP runs ahead of the CP. However, in the basic HiDISC design, the instruction stream for the CMP is quite similar to the Access Stream, which is a significant limitation as far as the effectiveness of the prefetching is concerned. Our original design executes every load instruction on CMP. However, if the cache line already resides in cache, those prefetches become redundant operations.

Only future probable miss instructions can benefit from the prefetches by the CMP. However, the current CMP is too heavy and involves performing too many redundant operations. Hence, in order to prefetch more efficiently into the cache, we must develop better methods so that we execute only probable miss instructions.

We define Cache Miss Access Slice (CMAS), which is a part of the Access Stream, consisting of the probable cache miss instruction and its parent instructions. The probable cache miss instructions can be found using the cache access profile [27][28]. The CMAS is executed on existing CMP in a multithreaded manner. Indeed, the CMP is an auxiliary processor for speculative execution of probable cache miss instructions.

5.2 Flexi-DISC

One of the most striking characteristics of the HiDISC architecture is its inherent flexibility and how it yields highly efficient execution of a large variety of loop-based programs with little or no temporal locality. This fundamental feature is further extended in the proposed Flexi-DISC. This new architecture will be targeted to a wide variety of more complex, numerical and non-numerical applications (such as Automatic Target Recognition).

While the original HiDISC is centered around three processors with well defined roles, the Flexi-DISC maintains the three roles of the CP, the AP, and the CMP at the kernel of its fundamental machine model but elevates it to a more sophisticated concept: the two highest levels (Access and Cache Management) are still handling the transfer of data between the memory system and the Computation level while the third level remains in charge of the computation per se. This can be represented as the three concentric rings on Figure 11: the Computation Kernel (CK), the Low-level Cache Access Ring (LCAR), and the Memory Interface Ring (MIR).

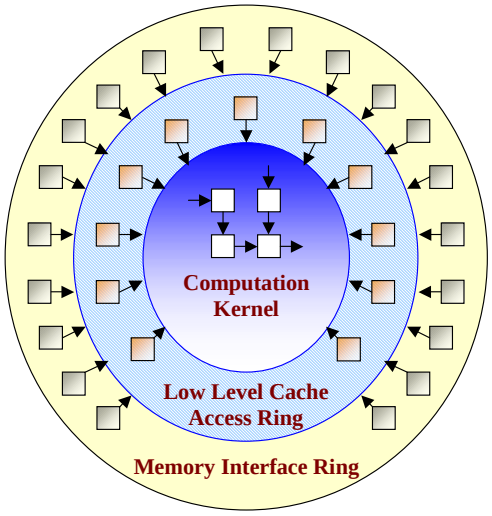

Figure 11: The three-Ring Flexi-DISC Architecture

The fundamental observation which leads to this partitioning comes from the fact that the types of applications (Memory Intensive) we have been targeting are both varied in nature and also inherently highly dynamic at execution time. This may mean that memory access patterns could range from, say, single use of any data element (no temporal locality), to multiple reuses (high temporal locality). Consequently, the bandwidth and types of pipes to and from the memory system must adapt to the changes, whether they be static or dynamic. We plan on centering the whole architecture around a highly reconfigurable Computation Kernel.

The central Computation Kernel is based on an array of simple processors which can be dynamically rearranged to meet the demands of the current application. It can even be partitioned into sub-arrays which are allocated to different portions of the application (or even to different applications as needed). Such a powerful computation kernel requires an equally powerful "pipeline" to feed it information to and from the memory system. Further, the variety of target applications makes the memory accesses unpredictable. This means that depending on the application (or even the phase of a given computation), the amount of memory traffic may fluctuate, and the prefetching mechanisms must be allowed to adapt to the situation at hand. This also means that

instead of allowing a single processor for the Cache Access role and another for the Cache Management role, a *pool* of identical processing units must be made available to the two roles combined. This sharing enables a highly efficient dynamic partitioning of the resources and their run-time allocation to the two outer rings (the Low-level Cache Access Ring, and the Memory Interface Ring).

The technology developed for the HiDISC compiler can be expanded to include the rearrange ability of the machine, as well as the partitioning it will undergo in the presence of multi-headed applications.

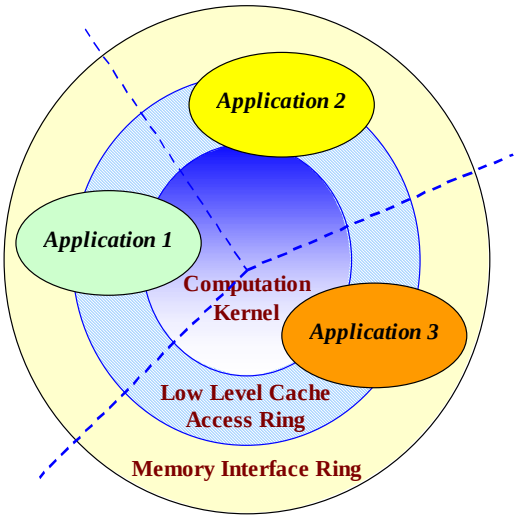

Figure 12: Multiple application sharing of the Flexi-DISC model

References

[1] Murali Annavaram, Jignesh M. Patel, Edward S. Davidson, Data Prefetching by Dependence Graph Precoumputation, *28th International Symposium on Computer* Architecture, June, 2001

[2] J.Arul, Execution Performance of the Scheduled Dataflow Architecture(SDF), International Conference on Parallel Architecture and Compilation Techniques: *MEDEA Workshop* Oct 13-15th 2000.

[3] A. Bakshi, Jean-Luc Gaudiot, Wen-Yen Lin, M. Makhija, V. K. Prasanna, Wonwoo Ro, Chulho Shin , Memory Latency: to Tolerate or to Reduce?, *The 12th Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2000* Oct 24-27, 2000

[4] P. Bird, A. Rawsthorne, and N. Topham. The effectiveness of decoupling. In *Int. Conf. on* Supercomputing, pages 47--56, 1993

[5] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *Technical report,* University of Wisconsin-Madison, Computer Science Department, 1997

[6] T.-F. Chen and J.-L. Baer. Effective Hardware-Based Data Prefetching for High-Performance Processors. *Transactions on Computers*, 44(5):609--623, May 1995

[7] Stephen P. Crago, HiDISC: a high-performance hierarchical, decoupled computer architecture, , *Ph.D. Dissertation,* University of Southern California, 1997

[8] Stephen P. Crago, Alvin Despain, Jean-Luc Gaudiot, Manil Makhija, Wonwoo Ro, and Apoorv Srivastava, A High-Performance, Hierarchical Decoupled Architecture, *In Proceedings of MEDEA Workshop, Philadelphia, October 15*, 2000

[9] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, John P. Shen, Speculative Precomputation: Long-range Prefetching of Delinquent Loads, *28th International Symposium on Computer* Architecture, June, 2001

[10] Haitao Du, Analysis of Memory Access Behavior of DIS Stressmark Suite and Optimization, *Tech . Report,* Center for Embedded Computer Systems, University of California, Irvine

[11] S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm and D. Tullsen, Simultaneous Multithreading: A Platform for Next-generation Processors, In *IEEE Micro*, 0. 12-19, Oct. 1997

[12] M. Farrens, P. Nico and P. Ng, A Comparison of Superscalar and Decoupled Access/Execute Architectures, In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, Dec. 1993

[13] J. R. Goodman, J. T. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young, PIPE: A VLSI Decoupled Architecture, In *Proc. 12th International Symposium on Computer Architecture*, pp. 20-27, June 1985

[14] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1992

[15] S.I. Hong, S.A. McKee, M.H. Salinas, R.H. Klenke, J.H. Aylor and W.A. Wulf, Access Order and Effective Bandwidth for Streams on a Direct Rambus Memory, *5th International Symposium on High-performance Computer Architecture*, 1999.

[16] G. P. Jones and N. P. Topham, A Comparison of Data Prefetching on an Access Decoupled and Superscalar Machine, In *Proc. 30th International Symposium on Microarchitecture*, Dec. 1997

[17] K.M. Kavi, J.Arul and R.Giorgi. Execution and cache performance of the Scheduled Dataflow Architecture, *Journal of Universal Computer Science, Special Issue on Multithreaded and Chip Multiprocessors*, Oct. 2000, pp 948-967, Vol. 6, No. 10.

[18] Ronny Krashinsky and Mike Sung, Decoupled Architectures for Complexity-Effective General Purpose Processors, *Tech. Report*, MIT Laboratory for Computer Science, Dec. 2000

[19] Lizy Kurian, Paul T. Hulina and Lee D. Coraor, Memory Latency Effects in Decoupled Architectures, *IEEE Transactions on Computers*, vol. 43, no. 10, Oct. 1994

[20] C.-K. Luk and T. C. Mowry. Compiler based prefetching for recursive data structures. In 7th *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996

[21] Chi-Keung Luk, Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processor, *In Proc. of International Symposium on Computer Architecture, 2001*

[22] Joan-Manuel Parcerisa and Antonio González, The Synergy of Multithreading and Access/Execute Decoupling, In Proc. *5th. Int. Symp. on High-Performance Computer Architecture (HPCA-5),* Jan. 1998

[23] Joan-Manuel Parcerisa and Antonio González, Improving Latency Tolerance of Multithreading through Decoupling, *IEEE Transactions on Computers*, October, 2001

[24] D. Patterson, et al. A Case for Intelligent DRAM: IRAM, *IEEE Micro*, April 1997

[25] Kevin Rich, Decoupled Architectures: A Thorough Analysis, Computer Science Department: *Qualifying Examination Paper*, University of California at Davis, Davis, California (May 1995)

[26] Amir Roth, Andreas Moshovos and Guri S. Sohi, Dependence Based Prefetching for Linked Data Structures. In *proc. ASPLOS-8*, October 4-7, 1998.

[27] Amir Roth and Gurindar S. Sohi, Speculative Data-Driven Multithreading, *In proc. of HPCA-7*, Jan. 2001

[28] Amir Roth, Craig B. Zilles and Gurindar S. Sohi, Speculative Miss/Execute Decoupling. *In proc. of MEDEA Workshop*, Oct. 19, 2000

[29] Jurij Silc, Borut Robic and Theo Ungerer, *Processor Architecture*, Springer, 1999

[30] J. Smith. Decoupled Access/Execute Computer Architecture. In *Proc. 9th International Symposium on Computer Architecture*, Jul. 1982

[31] Srikanth T. Srinivasan and Alvin R. Lebeck, Load latency tolerance in dynamically scheduled processors, In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, 1998.

[32] D. M. Tullsen, S. J. Eggers, and H.M. Levy, Simultaneous Multithreading: Maximizing On-Chip Parallelism. *In Proc. 22nd International Symposium on Computer Architecture* , June 1995.

[33] G. Tyson, M. Farrens and A. Pleszkun, MISC: A Multiple Instruction Stream Computer, *Proceedings of the 25th Annual International Symposium on Microarchitecture*, Dec. 1992

[34] Wm. A. Wulf, Evaluation of the WM Architecture, In *Proc. 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992

[35] Yinong Zhang, G. B. Adams III, Performance Modeling and Code Partitioning for the DS Architecture, In *Proc. 25th Annual International Symposium on Computer Architecture*, 1998

[36] Yinong Zhang, G. B. Adams III, Exploiting Instruction Level Parallelism with the DS Architecture. In *Proc. of the 1996 Int'l Conf. on Parallel Processing*, Aug. 1996

[37] Craig B. Zilles and Gurindar S. Sohi, Understanding the Backward Slices of Performance Degrading Instructions. *ISCA-2000*, June 2000.

[38] DIS Stressmark Suite, http://www.aaec.com/projectweb/dis/DIS_Stressmarks_v1_0.pdf

[39] Data-Intensive Systems Benchmarks Suite Analysis and Specification , http://www.aaec.com/projectweb/dis/

Appendix A: Compiler and Simulator Description

The compiler and the simulator are based on the SimpleScalar 3.0 tool set. The two tools have been designed by modifying *sim-outorder.c*. The first tool is *sim-pfg.c*, which takes care of the whole compiling procedure and the other one is *sim-dumas.c*, which exactly matches the HiDISC simulator. This appendix gives a detailed description of the tools.

A.1. Compiler Tool: *sim-pfg.c*

sim-pfg.c is the source code (C) for the HiDISC compiler. The main tasks of *sim-pfg.c* are: 1. Deriving the Program Flow Graph and 2. Separating the streams. The input for *sim-pfg.c* is a binary executable for SimpleScalar while the output is a binary executable for the HiDISC architecture with the separation information.

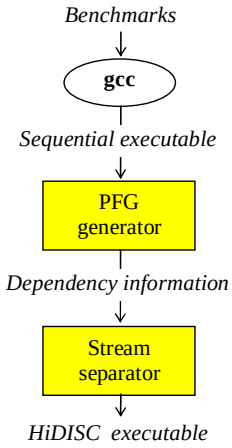

Figure 13: The HiDISC Compiler

Figure 13 shows the procedure inside the HiDISC compiler. The two boxes perform the operations mentioned earlier.

Deriving Program Flow Graph (PFG)

The Program Flow Graph delivers the data dependency information between instructions. The dataflow relationship between instructions must first be defined in order to get the

backward slice of a certain target instruction. After this procedure, each access related instruction can point to the parent instructions based on the source register name. The main procedure is named *pfg_const( )*. Its detailed mechanism is described in Figure 14.

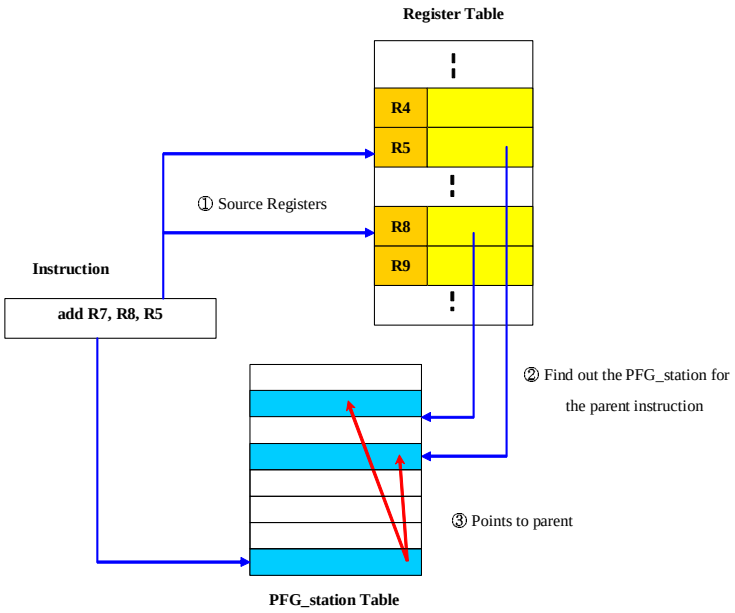

Figure 14: Deriving PFG Graph

The data structure for each instruction has been defined as *pfg_station*. After the instruction is decoded, a dedicated *pfg_station* is assigned. The first procedure consists in accessing the register table based on the source register name. (referred to as ① in Figure 14). The register table gives the pointer to the instruction (actually, the pointer to *pfg_station* of the instruction, referred to as ② in Figure 14) which last updated the source register. Finally, the decoded instruction can have the pointer for the parent instructions referred to as ③ in Figure 14.

This is how we uncover the parent instructions of a load/store instruction. Later, we can proceed with a backward chasing procedure in order to extract the backward slice based on the PFG information.

Separating Stream

The stream separation is based on the register dependencies. First, when the decoded instruction is either a load or a store instruction, it is immediately assigned to the Access Stream. After that, the backward chasing procedure is initialized (procedure named *chasing_parents( )* is called). Essentially, it is function call which is recursively applied until it reaches an instruction which has been pre-determined to belong to the Access Stream.

The PFG information from the previous step yields the pointers to the parent instructions. Therefore, the *chasing_parents( )* procedure basically returns all the pointers to the parent instructions.

After the instruction is detected as belonging to the Access Stream, the stream separation information is updated inside the binary file. Since each instruction of the SimpleScalar binary includes an additional annotation field, those extra bits can be used to carry the separation information.

A.2. Simulator: *sim-dumas.c*

The HiDISC simulator has been designed by modifying the *sim-outorder.c* module of the SimpleScalar 3.0 tool set [5]. The major modifications consist in: 1. implementing the three processors of the HiDISC and 2. implementing the communication mechanisms (queues) between those three processors. As in the original SimpleScalar simulator, the HiDISC simulator is also an execution-driven, cycle- time simulator.

To implement the three processors of the HiDISC, we basically copied three times the pipelined RISC processor of the SimpleScalar tool set and tailored each so they would correspond to the architecture of each HiDISC processor.

After the decoding stage, each processor has a corresponding *ready list*, which is the instruction stream for each processor. We implement three different functional units which are unique to each processor. Procedure *ruu_issue( )* of the *sim-outorder.c* has been copied and changed to *ruu_issup_cp( ), ruu_issue_ap( ),* and *ruu_issue_cmp( ).*

Each function detects each *ready list* and finds the available functional unit that is assigned to the corresponding processor.

The need for communication can also be detected at the decoding stage. If an instruction requires data from the other processor, it should be blocked and it should wait until the other processor sends the data. The queue implementation is quite easily handled using the existing link operations of the SimpleScalar tool set. All the necessary source data is linked after the *ruu_dispatch( )* procedure. Therefore, the sending processor can "wake up" the waiting processor just like *ruu station* in *sim-outorder.c*.

Communications between the AP and the CMP are achieved through the data cache. Therefore, the data cache is designed and implemented to be shared and accessed by both processors.

Appendix B: Raw Performance Data

This appendix contains all the simulation results. The column denoted as *mem* corresponds to the various memory latencies. The column marked SS contains the performance of the base line superscalar architectures. The fourth column denoted as HiDISC contains the performance results of the HiDISC architecture without the CMP processor. The remaining two contain the performance results with the CMP enhanced pre-fetching algorithms. The performance measures are all in IPC (instructions per clock).

< DIS benchmarks >

FFT	mem	SS	HiDISC	cmp1	cmp2
	20	1.2	1.56	1.79	1.83
	40	0.96	1.21	1.37	1.44
	60	0.79	0.99	1.12	1.18
	80	0.68	0.84	0.94	1.01
	100	0.59	0.72	0.81	0.87
	120	0.53	0.64	0.71	0.77

MoM	mem	SS	HiDISC	cmp1	Cmp2
	20	2.09	2.02	2.02	2.02
	40	2.06	2.02	2.02	2.02
	60	2.03	2.02	2.02	2.02
	80	2.01	2.02	2.02	2.02
	100	1.98	2.02	2.02	2.02
	120	1.95	2.02	2.02	2.02

DM	mem	SS	HiDISC	cmp1	cmp2
	20	1.3	1.47	1.47	1.48
	40	1.29	1.45	1.45	1.46
	60	1.27	1.43	1.43	1.45
	80	1.25	1.41	1.42	1.43
	100	1.23	1.39	1.4	1.41
	120	1.22	1.38	1.38	1.4

RayTracing	mem	SS	HiDISC	cmp1	cmp2
	20	1.0981	1.4633	1.4633	1.4633
	40	1.0966	1.463	1.463	1.463
	60	1.0951	1.4627	1.4628	1.4628
	80	1.0936	1.4624	1.4625	1.4625
	100	1.0922	1.462	1.4623	1.4623
	120	1.0907	1.4627	1.462	1.462

< Stressmark >

Transitive Closure	mem	SS	HiDISC	Cmp1	cmp2
	20	1.02	0.94	0.98	0.98
	40	0.63	0.91	0.98	0.95
	60	0.46	0.87	0.98	0.9
	80	0.36	0.86	0.98	0.89
	100	0.29	0.83	0.98	0.86
	120	0.25	0.82	0.98	0.83

Pointer	mem	SS	HiDISC	cmp1	cmp2
	20	1.33	1.46	1.46	1.46
	40	1.33	1.46	1.46	1.46
	60	1.32	1.46	1.46	1.46
	80	1.32	1.46	1.46	1.46
	100	1.32	1.45	1.46	1.46
	120	1.32	1.45	1.45	1.46

Neighbor-hood	mem	SS	HiDISC	cmp1	cmp2
	20	1.9	1.36	1.36	1.36
	40	1.9	1.36	1.36	1.36
	60	1.9	1.36	1.36	1.36
	80	1.9	1.36	1.36	1.36
	100	1.9	1.36	1.36	1.36
	120	1.9	1.36	1.36	1.36

Update	mem	SS	HiDISC	cmp1	cmp2
	20	1.676	1.5225	1.5225	1.5225
	40	1.6759	1.5225	1.5223	1.5223
	60	1.6759	1.5225	1.522	1.5221
	80	1.6759	1.5225	1.5218	1.5218
	100	1.6758	1.5224	1.5216	1.5216
	120	1.6758	1.5224	1.5214	1.5214

Matrix	mem	SS	HiDISC	cmp1	cmp2
	20	1.5131	0.8128	0.85	0.87
	40	1.3527	0.7832	0.8344	0.8542
	60	1.2224	0.738	0.8115	0.841
	80	1.115	0.6953	0.7867	0.8249
	100	1.025	0.6583	0.7649	0.8109
	120	0.9484	0.6246	0.7435	0.7966

Field	mem	SS	HiDISC	cmp1	cmp2
	20	1.75	1.6	1.6	1.6
	40	1.75	1.6	1.6	1.6
	60	1.75	1.6	1.6	1.6
	80	1.75	1.6	1.6	1.6
	100	1.75	1.6	1.6	1.6
	120	1.75	1.6	1.6	1.6