HiDISC: A Decoupled Architecture for Data-Intensive Applications

Won W. Ro, Jean-Luc Gaudiot†, Stephen P. Crago‡, and Alvin M. Despain

Department of Electrical Engineering
University of Southern California
{wro, despain}@usc.edu

†Department of Electrical Engineering and Computer Science
University of California, Irvine
gaudiot@uci.edu

‡ Information Sciences Institute-East
University of Southern California
crago@isi.edu

Abstract

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

Our performance evaluation benchmarks include the Data-Intensive Systems Benchmark suite and the DIS Stressmark suite. Our simulation results point to a distinct advantage of the HiDISC system over current prevailing superscalar architectures for both sets of the benchmarks. On the average, a 12% improvement in performance is achieved while 17% of cache misses are eliminated.

1. Introduction

The speed mismatch between processor and main memory has been a major performance bottleneck in modern processor architectures. While processor speed has been improving at a rate of 60% per year during the last decade, access latency to the main memory has been improving at a rate of less than 10% per year [10]. This speed mismatch – the Memory Wall problem - results in a considerable cost in terms of cache misses and severely degrades the ultimate processor performance. The problem becomes even more acute when faced with the highly data-intensive applications which are becoming more prevalent. By definition, they have a higher memory access/computation ratio than “conventional” applications.

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, databases, embedded processors, automatic target recognition, and any other data-intensive programs exhibit irregular memory access patterns [7] and result in considerable numbers of cache misses with 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 (and even further to register) before it is even requested by the CPU. Hardware prefetching [3] dynamically adapts to the runtime memory access behavior and decides the next cache block to prefetch. Software prefetching [9] usually inserts the prefetching instructions inside the code. Although previous research on prefetching considerably contributed to improvements in cache performance, prefetching techniques still suffer when faced with irregular memory access patterns. It is quite impossible to make valid predictions when the access patterns are nearly random [8]. Moreover, many current data-intensive applications use sophisticated data structures with pointers which dramatically lower the regularity of memory accesses.

The goal of this paper is to introduce a new architecture called HiDISC (Hierarchical Decoupled Instruction Stream Computer) [4] specifically designed for data-intensive applications. Two distinct contributions of the research are:

- Performance evaluation of data-intensive applications using an access/execute decoupled model: previous decoupled architecture research has focused only on scientific applications. However, we will demonstrate how data-intensive applications fit decoupled architectures.

- The introduction of hierarchical data prefetching using two dedicated processors: one additional processor has been designed and simulated on an existing access/execute decoupled architecture model, with an emphasis on cache prefetching.

The paper is organized as follows: In Section 2, we give the background research to the proposed architecture. Section 3 presents the detailed hardware description of the HiDISC architecture. In Section 4, the software aspect of the architecture will be presented. Section 5 includes experimental results and analysis. Conclusions and future research directions are included in Section 6.
2. Background research

The HiDISC architecture actually borrows from a number of established, related research domains, including access/execute decoupled architectures and speculative data prefetching.

2.1. Access/execute decoupled architectures

Access/execute decoupled architectures have been developed to exploit the parallelism between memory access operations (data fetch) and other computations. Concurrency is achieved by separating the original, single instruction stream into two streams - the Access Stream and the Execute Stream - based on the memory access functionality. Asynchronous operation of the streams provides for a certain distance between the streams and theoretically renders “perfect” data prefetching possible. Our HiDISC architecture is an enhanced variation of conventional decoupled architectures.

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 important parameter will be the “distance” between the instruction in the Access Stream currently producing a data and the instruction in the Execute Stream which uses it. 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 and they guarantee the correctness of program flow).

Early decoupled architectures were first introduced in the mid-80s [14]. PIPE [6] and the Astronautics ZS-1 [15] were the first generation of decoupled architectures. Although decoupled architectures have important features which allow them to tolerate long access latencies, actual implementation and development of the architectures have disappeared over the past decade, mainly because of the lack of software compatibility. However, the memory latency has become a problem more serious than ever and access decoupling should be considered again as one possible remedy.

2.2. Speculative data prefetching

One newly introduced prefetching method, which uses speculative execution of access-related instructions, is motivated by the introduction of decoupled architecture principles [12]. Instead of guessing future memory accesses, with this approach, we would pre-execute the necessary slice for future probable cache miss instructions. Speculative data-driven multithreading (DDMT) [12][13] is one of those approaches. Miss Streams called performance degrading slices or data-driven threads (DDT) include the cache miss instructions and backward slices. In Roth and Sohi’s approach [12][13], the instructions in the miss stream are executed in a multithreaded fashion for data prefetching.

The concept of performance degrading slice was originally introduced by Zilles and Sohi in [16]. In it, the cache miss instructions and branch misprediction instructions are chased for the backward slice. The analysis is based on the cache-profiling of the binary code and the use of a number of techniques to reduce the size of performance-degrading slices. The authors conclude that only a small part (less than 10%) of the instruction window is used by the cache misses or mispredictions. This work offers a proof of feasibility and a motivation for the use of pre-execution of performance-degrading threads [12].

DDMT implementation on the Itanium processor is described in [5], which utilizes the SMT feature of the Itanium processor for speculative data prefetching. Notably, they introduced the chaining trigger mechanism which allows speculative threads to trigger other speculative threads. Another approach using hardware for prefetching threads was introduced by Annavaram in [1]. A Dependence Graph Precomputation scheme (DGP) dynamically uncovers the parent instructions of cache miss instructions. In the DGP scheme, the speculative prefetching slices run on a pre-computation engine and only update the cache status.

3. The HiDISC design

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 manage the memory hierarchy and respectively prefetch the data stream.

3.1. Overall architecture of the HiDISC system

![Figure 1: The HiDISC system](image)

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 includes one more processor for data prefetching [4] (Figure 1). A dedicated processor for each level of the memory hierarchy performs a timely supply of 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:

- **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. The separation of the CP and the AP is quite similar to the traditional access/execute decoupled architectures. Our architectural innovation is located in the design and implementation of the CMP processor.

The main motivation for the existence of the CMP processor is to reduce the cache miss rate of the Access Processor by timely prefetching. Therefore, the CMP should run ahead of the AP, just like the AP runs ahead of the CP. Although the decoupled architectures provide an effective mechanism for data prefetching, the increasing memory access latency has imposed much burden on the AP. The AP has to be stalled at each cache miss and cannot provide data in time. With the introduction of the CMP, the cache miss caused on the AP can be reduced. The probable cache miss instructions and parent instructions are executed on the CMP, which executes a smaller amount of code and therefore can run faster than the AP. For that purpose, we define a Cache Miss Access Slice (CMAS), which is a part of the Access Stream, consisting of the “probable” cache miss instructions and backward slice.

We use a cache access profile to detect probable cache miss instructions. A similar approach is taken by previous speculative data prefetching research [1][5][12][13]. The CMAS is executed on the CMP in a multithreaded manner. Indeed, the CMP is an auxiliary processor for speculative execution of probable cache miss instructions. In other words, the CMP is very loosely coupled with the processors above it, because data communication occurs mainly through the cache.

Figure 2 shows the inside structure of the HiDISC architecture. The separator decodes the annotation field to decide the stream to which the instruction belongs. After the separation stage, the computation instructions are sent to the Computation Instruction Queue and the access instructions are sent to the Access Instruction Queue. The Computation Instruction Queue provides the necessary slip distance between the CP and the AP. After the separation stage, each instruction queue for each processor is filled with its respective stream of instructions. Each instruction decoding (ID) unit receives instructions only from the dedicated instruction queue and performs the decoding.

The CMAS stream is triggered when a trigger instruction is detected at the Access Instruction Queue. Then, the context of the Access Processor is forked to the CMP and the trigger instruction and the following CMAS instruction will be fetched to the CMP processor. The Access Instruction Queue provides the slip distance between the AP and the CMP.

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. As an example, Figure 3 shows the stream separation for the inner loop of the discrete convolution algorithm in C. The three instruction streams, which would executed on the HiDISC are shown in C-based pseudo-code.

Figure 3: Discrete Convolution as processed by the HiDISC compiler

The control flow instructions are executed by the AP. Incidentally, it should be noted that additional instructions are required in order to facilitate the synchronizations 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 loops.

3.2. Communication structure

The Load Data Queue (LDQ) and the Store Data Queue (SDQ) facilitate communication between the CP and the AP. The LDQ is designed to serve two purposes. First, the queue transmits data between the AP and the CP. The LDQ takes data which has been fetched or computed by the AP and passes it on to the CP. The second purpose of the LDQ is to provide synchronizations between the CP and the AP. Synchronization points are provided by requiring the CP to stall when it tries to read from an empty queue.

Likewise, the CP sends data, which will be stored by the AP, to the SDQ. The AP writes the memory address of the
data in the Store Address Queue (SAQ). When both the SDQ and the SAQ are non-empty, a store operation is executed and the data is sent to the data cache.

4. Operation of the HiDISC compiler

To drive the proposed HiDISC architecture effectively, the proper separation of each stream is extremely crucial. This section describes the stream separation function of the HiDISC compiler.

4.1. Evaluation 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 [2] 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 timing delays.

The HiDISC simulator has been designed by modifying the sim-outorder.c module of the SimpleScalar 3.0 tool set [2]. 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 instruction level, cycle-time simulator.

Each benchmark program follows the two steps. The first step is compiling the target benchmark using the HiDISC compiler which we have designed, while the second step is the simulation and performance evaluation phase. After the first step, each instruction has its separation information on the annotation field of the SimpleScalar binary. Then, the HiDISC simulator separates each stream at decoding stage according to the annotation field. The next section gives more detailed description of the first step.

4.2. Stream separation: Program slicing

The core operation of the HiDISC compiler is stream separation. It is achieved by a backward chasing of load/store instructions based on the register dependencies. In order to obtain the register dependencies between instructions, a Program Flow Graph (PFG) must be derived. Therefore, the PFG generation and the stream separation are two major operations of the HiDISC compiler. Both tools are adopted after some modifications from the SimpleScalar 3.0 tool set and integrated in the HiDISC compiler. Figure 4 depicts the flow of the HiDISC compiler.

The input to the HiDISC compiler is a conventional sequential binary code. The first step (1: Deriving the Program Flow Graph) 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 parent instructions. 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.

4.3. Insert Communication Instructions

After separating two streams, the third instruction stream named CMAS (Cache Miss Access Slice) should be prepared for the CMP. Basically, the CMAS is a subset of the Access Stream and constructed for the effective prefetching of data. The CMAS is defined with the help of the cache access profile. It consists of probable cache miss instructions and
backward slices. The CMAS is triggered in the CMP in a multithreaded manner. This means that the CMP is designed for speculative execution of the future memory access [12][13][16]. In our HiDISC architecture, we implemented an instruction window 512 in size so as to detect the trigger instruction. The additional annotation field of each instruction is used to convey the CMAS as well as the trigger information.

Indeed, the execution of the CMAS is an auxiliary operation. It only updates the cache status. Therefore, the program status is not affected by the CMP operation. To achieve timely prefetching and also prevent cache pollution, the prefetching distance should be defined properly. Currently, the instruction which is 512 instructions away from the cache miss instruction is defined as a trigger instruction. When the AP sees the trigger instruction, the CMAS is forked from the AS. More research will be conducted as future work on determining the trigger instruction and prefetching distance.

Figure 5, 6 and 7 are presented to describe how the HiDISC compiler operates. First, Figure 5 shows an example of the operation of the backward slicing mechanism. The example is shown as MIPS assembly code. The actual input code to the HiDISC compiler is the PISA (Portable Instruction Set Architecture) which is the instruction set of the SimpleScalar simulator [2]. The PISA code is compiled into SimpleScalar binary by first using a version of gcc which targets SimpleScalar. We have selected for this example the inner product of Livermore loop (III).

![Figure 5: Backward chasing of load/store instructions](image)

Initially, each memory access instruction is defined as belonging to the Access Stream. For example, the `lw` instruction in the fifth line (pointed to by an arrow  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 `mul` instruction in the fourth line (pointed to by an arrow  ) - due to the register $9$ - and the `addu` instruction in the second line (pointed to by an arrow  ) - 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 5 belong to the Access Stream.

After defining each stream, the communication instructions should be inserted. The solid lines with a forward arrow (on the right side of the instruction stream) in Figure 5 show the necessary communications from the Access Stream to the Computation Stream. For example, the `mul` instruction (which is marked as being inside the Computation Stream, pointed to by arrow  ) in the seventh line requires data from the other instruction stream (The Access Stream). Therefore, both `ld` instructions in the fifth and sixth line need to send data to `LDQ`. Likewise, the dotted line with a forward arrow at the bottom also shows the communication from the CS to the AS via the Store Data Queue (SDQ). Figure 6 shows the complete separation of the two streams and insertion of the communication instructions.

![Figure 6: Separation of sequential code](image)

After separating the code into two streams, the code for the CMP (named as CMAS) is basically derived from the Access Stream. The CMAS code can be identified by using the cache miss profile. For example, if the instruction pointed to by arrow  in Figure 7 has been detected as a cache miss instruction by using the access profile, it is defined as a probable cache miss instruction and immediately marked as belonging to the CMAS. Then, the parent instructions are chased backward and also defined as belonging to the CMAS. In Figure 7, `addu` (pointed to by arrow  , due to `$11$) and `la` (pointed to by arrow  , due to `$10$) are chased and defined as belonging to the CMAS. As explained above, we define a trigger instruction to execute CMAS using 512 instruction window from the probable cache miss instruction.

![Figure 7: Defining CMAS (cache miss access slice)](image)
5. Experimental results

We have based our research upon a deterministic simulation of the HiDISC architecture, as well as a comparison to a number of similar architectures in order to accurately identify the architectural improvements which yield the highest performance improvement. The simulation has focused on the simulated execution of a number of data intensive benchmarks. Our methodology, as well as the benchmark description is now outlined.

5.1. 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 been quite pronounced in a variety of environments [17]. Indeed, many applications such as Automatic Target Recognition (ATR) and database management show non-contiguous memory access patterns and often result in idle processors due to data starvation. These applications are more stream-based and result in more cache misses due to the lack of locality.

Frequent use of memory de-referencing and pointer chasing also causes increased 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 use of Object-Oriented Programming correspondingly increases the underlying use of pointers. Due to the serial nature of pointer processing, memory accesses become a severe performance bottleneck in 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 strongly dependent upon one another [11].

The applications for which our HiDISC is designed are obviously data-intensive programs, the performance of which is strongly affected by the memory latency. We used for our benchmarks the Data-Intensive Systems Benchmark suite [17] and the DIS Stressmark suite [18] designed by the Atlantic Aerospace Electronics Corporation. Both of the benchmarks are targeting data-intensive applications. The Data Intensive Systems (DIS) benchmark includes five benchmarks, which are more realistic and larger than Stressmark. Stressmark includes seven small data-intensive benchmarks, which extract and show the kernel operation of data-intensive programs.

5.2. Simulation parameters

In our benchmark simulations, the architectural parameters shown in Table 1 have been used. The baseline architecture for the comparison is an 8-issue superscalar architecture, which is implemented as sim-outorder in the SimpleScalar 3.0 tool set. It also supports out-of-order issue with 64 entries register update units and 32 entries load store queues. The size of scheduling window for the AP is also 64.

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch predict mode</td>
<td>Bimodal</td>
</tr>
<tr>
<td>Branch table size</td>
<td>2048</td>
</tr>
<tr>
<td>Issue/commit width</td>
<td>8</td>
</tr>
<tr>
<td>Instruction scheduling window size</td>
<td>Superscalar / DEA / HiDISC Access Processor: 64 /Execute Processor: 16</td>
</tr>
<tr>
<td>Integer functional units</td>
<td>ALU (x 4), MUL/DIV (for Superscalar, CP, AP, and CMP)</td>
</tr>
<tr>
<td>Floating point functional units</td>
<td>ALU (x 4), MUL/DIV (for Superscalar and CP)</td>
</tr>
<tr>
<td>Number of memory ports</td>
<td>2 for Superscalar / 2 for each processor (AP and CMP)</td>
</tr>
<tr>
<td>Data L1 cache configuration</td>
<td>256 sets, 32 block, 4-way set associative, LRU</td>
</tr>
<tr>
<td>Data L1 cache latency</td>
<td>1 CPU clock cycle</td>
</tr>
<tr>
<td>Unified L2 cache configuration</td>
<td>1024 sets, 64 block, 4-way set associative, LRU</td>
</tr>
<tr>
<td>L2 cache latency</td>
<td>12 CPU clock cycles</td>
</tr>
<tr>
<td>Memory access latency</td>
<td>120 CPU clock cycles</td>
</tr>
</tbody>
</table>

5.3. Benchmark results

Four architecture models including the baseline superscalar have been defined and simulated to evaluate the HiDISC architecture. The first model is the baseline superscalar architecture. The second model is a conventional access/execute decoupled architecture. Only the CP and the AP are activated for this configuration referred to as the CP+AP configuration. The third case is examined with running the CP and the CMP (referred to as the CP+CMP configuration). In this configuration, a single instruction stream (including the Computation Stream and the Access Stream) is fetched into the CP and only the CMAS is triggered to run on the CMP for cache prefetching. It is a model quite close to that of DDMT [13] and Speculative Pre-computation [5]. The fourth configuration is the complete HiDISC.

Figure 8 displays the performance improvement for each configuration compared to the baseline superscalar architecture. For each benchmark, the far left bar corresponds to the normalized performance result of the baseline architecture. The second bar expresses the performance result of the CP+AP configuration. The third bar shows the performance result of the CP+CMP configuration. The right most bar displays the performance result of the HiDISC architecture. In the six cases except the Neighborhood Stressmark, the HiDISC shows the best performance. The upper bound of speed-up is 18.5% which is achieved for the Update Stressmark. The average speed-up is 11.9% across all seven benchmarks.

In most cases, prefetching by the CMP is a major performance enhancement factor (since the two configurations with the CMP outperform the other two without the CMP). However, the Field Stressmark
eloquently shows the merit of the access/execute decoupling over the CMP. Indeed, the help of the CMP is not outstanding for this benchmark. It is because the Field Stressmark contains a relatively small number of cache misses and therefore cannot benefit much from the data prefetching.

On the contrary, the Neighborhood Stressmark and the Transitive Closure Stressmark do not show any performance enhancement from the access/execute decoupling. More notably, in Neighborhood, the CP+AP configuration experienced performance degrading compared to the baseline superscalar architecture. It also affects the performance of the HiDISC (the CP+CMP model performs better than the HiDISC only in the Neighborhood Stressmark). It is mainly due to that the frequent synchronizations between the CP and the AP cause stalling of the processors. In those cases, the CP loses many CPU cycles to wait until the necessary data arrives. This means the enough slip distance between two processors is not guaranteed. We call these events loss of decoupling events.

Table 2: Speed-up for the three architecture models

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Characteristic</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP + AP</td>
<td>Access/execute decoupling</td>
<td>1.3%</td>
</tr>
<tr>
<td>CP + CMP</td>
<td>Cache prefetching</td>
<td>10.7%</td>
</tr>
<tr>
<td>HiDISC</td>
<td>Decoupling and prefetching</td>
<td>11.9%</td>
</tr>
</tbody>
</table>

The average performance improvement of the three architectures compared to the baseline superscalar is shown in Table 2. The major performance enhancement factor can be found in the prefetching capability of the CMP. The performance enhancement of the CP+AP configuration is not remarkable, when compared to the superscalar architecture. This is because the large instruction window of the superscalar architecture also performs dynamic scheduling of the load instructions.

To analyze the effectiveness of the prefetching in more detail, the cache miss rate for each configuration has also been calculated. Figure 9 shows the reduction in the cache miss rate for each configuration as compared to the baseline architecture. As the results indicate, the number of cache misses is considerably reduced when the CMP operates. The best case is achieved with the Transitive Closure benchmark, where the cache misses have been reduced by a factor of 26.7%. On the average, 17.1% of the cache misses are eliminated by the HiDISC.

In order to demonstrate how well HiDISC would tolerate memory latency, we varied the memory latencies of HiDISC for the simulation of Pointer and Neighborhood (Figure 10). The longest latency configuration we considered is: (memory access latency = 160) and (L2 cache access latency = 16). The shortest case is (memory access latency = 40) and (L2 cache access latency = 4). In general, the two architecture models which include the CMP show distinct robustness with long latencies. However, the other two architectures suffer from longer latencies. Indeed, the CP+AP configuration also suffers even it has the access/execute decoupling feature. The access/execute decoupling mechanism cannot effectively cope with the long latencies because of the synchronizations between the two streams.

For the Pointer Stressmark benchmark, the HiDISC architecture shows only 1.8% of performance degradation for the longest latency as compared to the shortest case. The baseline superscalar loses upwards of 20.3% of performance at the longest latency. In the Neighborhood Stressmark, the performance degradation of the HiDISC at the longest latency is merely 4.8% (It is also compared to the shortest latency). The baseline architecture loses a whole 13.9% at the longest latency.

6. Conclusions

The effectiveness of the HiDISC architecture has been demonstrated with data-intensive applications. It has been eloquently shown that the proposed prefetching method provides better ILP compared to the conventional superscalar architectures. The HiDISC architecture contains two characteristics: the access/execute decoupling and the cache prefetching. To demonstrate the contribution of each characteristic, four architecture models including a baseline...
superscalar architecture have been defined and tested here. The results indicate that, in the HiDISC, decoupling provides less contribution to the ultimate performance than the prefetching ability. It is mainly due to the loss of decoupling events. Too many data dependencies of the access processor prevent a sufficient slip distance between two streams.

Future work could include adding dynamic control of the CMAS execution for effective prefetching. The runtime control of the prefetching distance is another important task. Especially under the various program behaviors and memory latencies, the prefetching distance should be selected dynamically. In addition to that, the range for prefetching can be controlled at runtime. This means that not every probable cache miss instruction would be triggered as CMAS. Depending on the previous prefetching history, we can choose only the necessary prefetching at run time.

![Figure 10: Latency tolerance for various memory latencies](image-url)