A hybrid partitioning and scheduling technique for branch decoupling

Pramod Bhanu Ramaro

Iowa State University
A hybrid partitioning and scheduling technique for branch decoupling

by

Pramod Bhanu Ramarao

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Akhilesh Tyagi, Major Professor
Zhao Zhang
Soma Chaudhuri

Iowa State University
Ames, Iowa
2003

Copyright © Pramod Bhanu Ramarao, 2003. All rights reserved.
Graduate College
Iowa State University

This is to certify that the master's thesis of

Pramod Bhanu Ramarao

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy
# TABLE OF CONTENTS

**LIST OF TABLES** .......................................................... vi

**LIST OF FIGURES** .................................................. vii

**ABSTRACT** ................................................................. ix

**CHAPTER 1. INTRODUCTION** ............................................. 1

1.1 Branches and Branch Resolution in Processors .................. 1

1.2 Branch Decoupled Architectures ................................. 3

1.3 Related Work ........................................................ 4

  1.3.1 Memory Decoupled Architectures .......................... 4

  1.3.2 Branch Decoupled Architectures .......................... 4

  1.3.3 Branch Prediction using Subordinate Microthreads ..... 6

  1.3.4 Thesis Contribution ........................................... 7

1.4 Thesis Organization .................................................. 7

**CHAPTER 2. BRANCH DECOUPLED ARCHITECTURES** ................. 8

2.1 Functioning Principle .............................................. 8

2.2 Hardware Architecture ........................................... 10

  2.2.1 The Branch Decoupled Processor .......................... 10

  2.2.2 Control Communication Mechanism ...................... 13

  2.2.3 Data Communication Mechanism ........................ 14

  2.2.4 Software Architecture ................................. 17

**CHAPTER 3. BRANCH DECOUPLING SIMULATION TOOL SET** ........ 21

3.1 Overview ............................................................. 21
3.1.1 The SimpleScalar Tool Set ........................................... 21
3.1.2 Methods for Performance Evaluation ............................... 23
3.1.3 Branch Decoupling Simulation Tool Set ............................ 24

3.2 An Out-of-order Execution-Driven Simulator ...................... 24
3.2.1 The Necessity for sim-bdcorder .................................. 26
3.2.2 Design of sim-bdcorder ............................................. 26

3.3 Sim-bdcorder Simulator ................................................ 27
3.3.1 The BDA Instruction Set ........................................... 27
3.3.2 The Program and Branch Processors ............................... 31
3.3.3 Architectural Queues of the BDA ................................. 32

3.4 Branch Decoupling Compiler .......................................... 33
3.4.1 The GNU C Compiler .............................................. 34
3.4.2 Register Transfer Language - GCC's IR .......................... 35
3.4.3 Control and Data Flow Analysis ................................. 35
3.4.4 Dependence Analysis using Alias Analysis ..................... 39

3.5 Branch Decoupling Algorithm ........................................ 39
3.5.1 Kernighan-Lin Algorithm ......................................... 40
3.5.2 Cost Function and Slack Analysis ............................... 43
3.5.3 Scheduling ......................................................... 45
3.5.4 Decoupling Process .............................................. 46
3.5.5 Generating Annotations .......................................... 49

3.6 Branch Decoupled Glibc and Binutils .............................. 51
3.6.1 Branch Decoupled glibc .......................................... 51
3.6.2 Branch Decoupling Binutils ..................................... 51

CHAPTER 4. SIMULATION AND RESULTS .................................... 52
4.1 Methodology ............................................................ 52
4.2 Benchmarks ............................................................ 52
4.3 Processor Configurations ............................................. 53
4.4 Results and Analysis .......................................................... 53
  4.4.1 Instruction Statistics ....................................................... 55
  4.4.2 Performance Statistics ................................................... 58

CHAPTER 5. CONCLUSION AND FUTURE WORK ......................... 62
  5.1 Conclusion ................................................................. 62
  5.2 Future Work ............................................................... 63
LIST OF TABLES

Table 3.1  SS Virtual memory space ................................................. 22
Table 3.2  BDA processor register definitions ............................... 31
Table 4.1  Benchmarks and input data sets .................................. 53
Table 4.2  Number of instructions executed and amount skipped ...... 54
Table 4.3  Architectural parameters ............................................ 54
Table 4.4  Branch instruction and memory reference counts ............ 56
Table 4.5  Distribution of instruction counts on each processor ...... 59
Table 4.6  Distribution of copy and sync instructions .................... 60
**LIST OF FIGURES**

| Figure 2.1  | A simple loop ................................................................. | 9 |
| Figure 2.2  | Assembly code for the loop .................................................. | 9 |
| Figure 2.3  | Block diagram of the BD processor architecture .......................... | 11 |
| Figure 2.4  | Source code to be decoupled ................................................ | 13 |
| Figure 2.5  | The decoupled streams ........................................................ | 14 |
| Figure 2.6  | Decoupling with implicit copies .............................................. | 16 |
| Figure 2.7  | BDA system software architecture ........................................... | 19 |
| Figure 2.8  | Application memory layout .................................................... | 20 |
| Figure 3.1  | SimpleScalar instruction encoding ........................................ | 22 |
| Figure 3.2  | Trace versus execution-driven simulators .................................. | 24 |
| Figure 3.3  | BDA evaluation methodology .................................................. | 25 |
| Figure 3.4  | Structure for an ru entry ..................................................... | 28 |
| Figure 3.5  | BDA instruction encoding ..................................................... | 29 |
| Figure 3.6  | Structure of the branch condition queue ................................... | 33 |
| Figure 3.7  | Structure of the sync queue .................................................. | 34 |
| Figure 3.8  | Terms used in KL algorithm .................................................. | 42 |
| Figure 3.9  | Partitioning using KL algorithm ............................................. | 43 |
| Figure 3.10 | Example of instruction slacks ................................................. | 45 |
| Figure 3.11 | Decoupling pass design and decoupling algorithm ........................ | 50 |
| Figure 4.1  | Ratio of instructions in the B and P stream ................................ | 57 |
| Figure 4.2  | IPC comparison between the base and BDA processors ..................... | 58 |
Figure 4.3  Availability of branch conditions when required  . . . . . . . . . . 61
Control hazards caused by conditional branches are one of the biggest obstacles to achieving performance in out-of-order superscalar processors. Branch prediction techniques help alleviate the penalties associated with branch instructions, but still exhibit mis-prediction rates due to their functioning principle. A new paradigm, Branch decoupled architectures, has been proposed as an alternative to reduce branch stalls. This paradigm supported by an accompanying compiler, has a two-execution-unit processor - a branch processor and a program processor. A program is decoupled during compile time into two instruction streams and executed on the branch decoupled processor. The objective of the decoupling process is to have the branch processor solve branch conditions and precompute branch target addresses in advance for the program processor. This thesis presents three contributions. An algorithm based on graph bi-partitioning and scheduling, used by the compiler for decoupling the program's instruction stream into two streams is presented. This technique attempts to achieve maximal decoupling and at the same time attempts to reduce interaction between the two streams. Maximal decoupling allows both processors to run as independently as possible thereby extracting maximum benefit from the branch decoupled architecture paradigm. Application of the decoupling algorithm has been shown to result in 48.6% and 38.1% of the instructions on the average being executed on the branch and program processors. Simulations show a performance improvement of 7.7% and 5.5% on the average for integer and floating point benchmarks respectively. It then presents a toolchain consisting of a compiler, binary utilities (assembler, linker, loader) and associated libraries that has been retargeted to the branch decoupled architecture platform. Finally an overview of an out-of-order execution-driven superscalar processor simulator that has been developed for simulating the branch decoupled architecture is presented.
CHAPTER 1. INTRODUCTION

This chapter describes the effects of conditional branches and problems associated with branch resolution in modern processors. An introduction to Branch Decoupled Architectures is provided and proposed as a solution to these problems. An overview of related work is then presented and the chapter concludes with a brief outline of the organization of the thesis.

1.1 Branches and Branch Resolution in Processors

Instructions that change the flow of control in a program are called branches. A significant fraction of the instructions executed in a general purpose application can be attributed to both conditional and unconditional branches [10]. For integer applications, in the SPEC2000 suite [25], 82% of the control instructions have been found to be conditional branch instructions and 18% of those are unconditional branches (calls/returns and jumps). Floating point applications have conditional branch and unconditional branch instructions accounting for 75% and 25% of the control instructions respectively [10].

Many contemporary scalar processors use pipelining to achieve high performance. Pipelining involves overlapping the execution of multiple instructions. Branches can account for a significant fraction of loss of performance in these processors. Conditional branch instructions can break the smooth flow of instruction fetching and execution as a subsequent instruction may be needed to be fetched from either the next sequential memory location or a non-sequential location called a branch target address (BTA). For conditional branches, the branch condition cannot be computed early enough in the pipeline to maintain high throughput. If the branch outcome is a taken result, the next instruction now needs to be fetched from the BTA instead of the next sequential location. This necessitates the pipeline to be flushed and leads to stalls.
in the pipeline called *Control Hazards.*

In addition to being pipelined, modern processors are also superscalar in nature. By allowing concurrent execution of instructions in the same pipeline stage, high performance and greater pipeline utilization is achieved. A wider issue bandwidth also allows multiple instructions to be issued and committed in each clock cycle. However, superscalar processors suffer from aggravated effects of control hazards. When the outcome of a branch is not known, the instruction fetch stage may begin to fetch from a predicted path. If this happens to be the incorrect path (known only when the branch outcome is actually computed in the later stages), the incorrect instructions need to be flushed from the pipeline. As a result, the pipeline needs to be stalled while the fetch is initiated in the correct direction. This leads to waste of cycles thereby decreasing the performance of a superscalar processor. For example, a wasted cycle in a 4-way processor, results in the loss of a potential four issues and four commits in that cycle.

The most widely used and effective technique used in modern processors to reduce branch penalties is speculative execution in conjunction with branch prediction. Dynamic hardware prediction using branch target buffers (BTB) allows caching of BTAs so that the BTAs are available much earlier in the pipeline. The processor instead of stalling on a conditional branch, speculatively continues fetching and executing instructions. The semantics of the program is preserved by not committing speculatively executed instructions until the prediction is validated. In case of a correct prediction, all the speculatively executed instructions are committed to the register file and no stalls are encountered. An incorrect prediction on the other hand, simply causes the pipeline to be flushed and restarted from the correct fetch direction. This technique however can lead to significant performance degradation through pipeline stalls if the instruction at the BTA causes a cache miss or if the branch is mis-predicted.

The effectiveness of speculative execution therefore relies on the accuracy of dynamic branch prediction. Techniques such as 2-level correlating predictors mis-predict at the rate of 1% to 18% [10]. This is due to their functioning principle and thus puts a limit on the achievable throughput that can be obtained from dynamic branch prediction.

Branches also affect instruction prefetching, which is a technique used to hide memory
latencies of the lower memory levels in a multi-level memory system. A conditional branch that is taken may invalidate the last prefetch into the instruction cache. If the instruction at the BTA results in a cache miss, then the pipeline would need to be stalled for several cycles until the new instruction is brought into the cache.

In order to reduce the effect of branches, many techniques have been proposed. Compiler techniques such as scheduling using delayed branch slot, loop unrolling, software pipelining [10], superblock and hyperblock scheduling [11][14] help alleviate the impact of branches and increase instruction level parallelism. Another effective technique is if-conversion where a branch instruction is converted into a instruction which computes guard predicates. Predicates are then used to decide whether an instruction needs be executed or not. This essentially converts a control dependence of an instruction on a branch condition to one of data dependence on the instruction that computes the predicates [14].

1.2 Branch Decoupled Architectures

In [26], Tyagi proposed Branch Decoupled Architectures (BDA) to help alleviate the branch resolution problem in processors. Instead of extracting marginal improvements from branch prediction techniques that are already very effective, the BDA attempts to provide a different approach to the branch resolution problem.

The computation backbone of a program can be visualized to be composed of two parts: a branch-related computation and the rest of computation. The branch-related computation resolves the branches and decides the control flow of the program. The rest of computation follows the established control flow and accomplishes the purpose of the program. Thus the instruction stream of a program can be divided into : a Branch Stream (B-Stream) and a Program Stream (P-Stream). The two streams are run on a Branch Decoupled Processor which is equipped with two execution units called Branch Processor and Program Processor designed to execute the two streams. The computation in the B-Stream is expected to run ahead of the P-Stream so that the branch outcomes are resolved before they are required by the P-Stream. Thus the Program Processor would experience no stalls for conditional branches.
In order to accomplish decoupling of branches from the instruction stream, dependence analysis needs to be performed. The purpose of dependence analysis is to allow branches and instructions that the branch depends on to be peeled from the main instruction stream. This requires extensive support from the compiler and hence the BDA is a compiler assisted architecture. The compiler performs dependence analysis, applies a decoupling algorithm to split the instruction stream and generates a two-stream executable for the BDA.

1.3 Related Work

1.3.1 Memory Decoupled Architectures

Decoupled access/execute architectures called Memory Decoupled Architectures (MDA) were introduced by Smith in [22] [23]. Here, decoupling is used to isolate instructions that are on the critical path of the execution of the program, long latency memory instructions in the case of MDA and execute them on separate execution units of the processor. Decoupling is achieved by splitting the instruction stream into two streams: instructions that access memory and those related to memory accesses are placed in one stream and the other instructions related to general computation are placed in the other stream. Separating instructions which access memory allows data to be prefetched so it's available to the other stream with minimum access latency. The MDA contains two processors: an address processor and an execution processor which communicate through architectural queues and have separate register files and instruction caches.

1.3.2 Branch Decoupled Architectures

Tyagi proposed the concept of branch decoupled architectures in [26] to alleviate branch effects in processors. However, decoupling in this case was applied to reduce control hazards rather than reducing memory access latencies. Haehre [9] implemented a prototype simulator for branch decoupled architectures. However, no support for copy instructions was provided for communication between the two streams and lacking a compiler and the necessary system software, only a limited set of programs was simulated. Two small programs were simulated,
one was the Lawrence Livermore Loop #1 and the other program was "sort.c". The results of simulations were encouraging and speedups of between 1.33 to 1.64 were achieved.

**Dynamic Branch Decoupled Architectures** In [27] Ng studied a different variant called *Dynamic Branch Decoupled Architectures*. Here, decoupling is performed at run-time and requires no compiler support. The advantage of this model is its compatibility to legacy programs. The dynamic branch decoupled processor dynamically decouples programs in a *Decoupling and Fetch Queue* and the resulting two streams are executed by two separate execution units in the processor. Simulations showed performance gains of up-to 40% over 2-level adaptive branch prediction for integer benchmarks [27].

**Branch Decoupling Compiler** A compiler supporting branch decoupled architectures was developed by Zhang in [28]. The simulations were performed using a scalar processor model instead of an out-of-order superscalar processor model. The preliminary studies showed that on the average 43% of the branches are resolved and available in a branch queue before they are required by the P-Processor. This is up to 81% for integer benchmarks. The decoupling algorithm used was simple and though it was not designed for instruction ratios, dynamic instruction counts indicated that the ratio between the B-Stream and P-Stream were good (40.5% of the instructions were in the B-Stream on the average). The compiler developed provided support for copy instructions to facilitate communication between streams. Since the BDA is a dual processor model, the data memory is shared between the two processors. This requires that memory accesses (both reads and writes) be properly synchronized between the processors so that memory accessed are always performed in the order intended in the program. Since synchronization points defeat the purpose of giving one stream a reasonable run-ahead, good memory alias analysis becomes important during compilation to distinguish between two memory accesses and synchronization points need to be inserted by the compiler only if necessary. However no memory alias analysis was incorporated in the compiler developed by Zhang and this resulted in too many unnecessary synchronization points between the two streams. Nonetheless, the important contribution of the work was the development of the necessary system software for the BDA [28].
Trace Based Branch Decoupling

Nadkarni [17] studied yet another variant where a trace-driven out-of-order simulator was developed to study the BDA. An execution trace is first generated by an initial run of the program by an execution-driven simulator. Simple decoupling algorithms are then applied on the execution trace and the generated trace is re-run on a trace-driven out-of-order simulator. Three variants of the decoupling algorithm were studied, one a simple decoupling algorithm, another which attempts to perform complete decoupling without copies and a third which tries to achieve a good balance of instructions between the two streams. Support for copy instructions and synchronization points are also provided. In order to enhance performance, branch prediction is used on both the branch and program processors. The results indicated a speedup of 1.14 to 1.17 in case of floating point benchmarks and a speedup of 1.12 to 1.46 in case of integer benchmarks [17].

1.3.3 Branch Prediction using Subordinate Microthreads

In [5], Patt proposed using Simultaneous Subordinate Microthreaded architectures to improve branch prediction. SSMT architectures spawn multiple concurrent microthreads in support of the primary thread and can be used for a variety of tasks. Subordinate threads are constructed dynamically and these speculatively pre-compute branch outcomes along frequently mis-directed paths. There is no software support and the entire mechanism is implemented completely in hardware. The idea is to identify difficult-paths that frequently mis-predict beyond a threshold to guide microthread prediction. When the mis-prediction threshold is exceeded for a particular control flow path, a microthread is dynamically constructed. The scope of a difficult-path are all the neighboring instructions that are guaranteed to execute each time the path is encountered. The dataflow within the scope is constructed and a subset of instructions that pre-compute the branch condition and target address is extracted. This becomes the micro-thread for that difficult-path. Hardware structures are used to guide difficult-path identification and microthread construction. Microthreads are carefully constructed since they compete with the main thread for resources and incur performance overhead for incorrect predictions. The construction process must guarantee that microthreads constructed complete
predictions in time otherwise they become useless. Performance gains of 8.4% on the average were obtained over a range of integer and floating point benchmarks.

1.3.4 Thesis Contribution

Although a compiler for the BDA was developed in [28], the decoupling algorithm used was a simple greedy-based algorithm. This thesis presents a decoupling algorithm based on graph bi-partitioning and scheduling that has been integrated into the compiler. The compiler performs decoupling based on this algorithm and generates two text-segment executables. It also presents a BDA simulation tool set that consists of a retargeted toolchain for the BDA. The toolchain consists of the branch decoupling compiler and the necessary binary utilities (linker, assembler) and libraries for creating two text-segment executables. The tool set also includes an execution-driven out-of-order simulator that executes the two text-segment binary executables thereby providing a simulation environment for the BDA.

1.4 Thesis Organization

The remainder of this thesis is divided into four chapters. Chapter 2 presents a detailed description of the hardware architecture of the BDA. It also provides an overview of the associated system software architecture required to support the BDA. Chapter 3 describes the Branch Decoupling Simulation Tool set, which is an environment for the simulation of the BDA. It provides details of the Branch Decoupling compiler, a decoupling algorithm based on graph bi-partitioning and an out-of-order execution simulator developed for simulating the BDA. The simulation methodology, results and analysis are discussed in Chapter 4. Chapter 5 concludes the thesis and provides an insight into future work in this direction.
CHAPTER 2. BRANCH DECOUPLED ARCHITECTURES

This chapter presents a more detailed overview of branch decoupled architectures.

2.1 Functioning Principle

A program is characterized by a sequence of instructions and the control flow paths along which the execution proceeds to achieve the desired purpose. This stream of instructions typically consist of a subset of instructions (which can be identified on control flow paths) that are responsible for computing the branch conditions and deciding the control flow directions of the execution. This subset of instructions can be thought of as Control Flow Deciding (CFD) instructions. The remaining instructions can be collectively called Main Computation Instructions (MCI).

The CFD instructions are the branch instructions and the instructions that the branch instructions have data dependences on. The main principle behind BDA toward alleviating branch related stalls is to decouple the CFD instructions from the program. The program is effectively divided into two instructions streams: branch stream (B-Stream) and the program stream (P-Stream). The two instruction streams are executed on the branch decoupled processor which has two execution units: branch processor (B-Processor) and program processor (P-Processor). The B-Processor executes the B-Stream and resolves the branch conditions so that they are available to the P-Processor which executes the main computation instructions of the program.

This can be illustrated with a simple loop as shown in Figure 2.1. The corresponding assembly translation is shown in Figure 2.2.

In the code segment, R1 is initially the address of the element in the array with the highest...
for ( i = 0; i < N; ++i )
    y[ i ] = y[ i ] + s;

Figure 2.1 A simple loop

address, and \( F2 \) contains the scalar value, \( s \). For simplicity, assume the element at the lowest address is at zero.

\[
\text{LOOP:} \quad \begin{align*}
    &\text{LD} \quad F0, \ 0(R1) \\
    &\text{ADDD} \quad F4, F0, F2 \\
    &\text{SD} \quad 0(R1), F4 \\
    &\text{SUBI} \quad R1, R1, \ #8 \\
    &\text{BNEZ} \quad R1, \text{LOOP}
\end{align*}
\]

Figure 2.2 Assembly code for the loop

In this loop, the last instruction which is the branch instruction and the instruction which computes the index \( R1 \) are the CFD instructions. The remaining instructions in the loop form the MCIs. This example also shows why inter-stream communication is important as the index value in \( R1 \) is required in both streams.

The principle of branch decoupling involves identifying the CFDs and the MCIs in the instruction stream. A branch decoupling compiler [28] is used to perform the decoupling and a two-stream binary executable is generated which is then executed by the BDA.

The decoupling process is imperative to extracting maximum benefit from the BDA and must have some important desirable characteristics.

- The B-stream should be lightweight. The instruction count should be smaller than that of the P-stream. This enables the B-processor to have a reasonable slip compared to the P-processor and allows it to precompute branch conditions before they are needed by the program processor. The control flow information is transferred from the B-processor to P-processor through architectural queues.
As shown in the previous example, computation results in one stream may be required by the other stream. Thus, either the instruction must be replicated in both streams or a copy must be inserted. Copy instructions allows a producer instruction in one stream to communicate the result to a consumer in the other stream. In order for both the streams to be as autonomous as possible, the number of copies should be minimized. This ensures that either stream is not stalled waiting for results from the other.

The Branch Decoupled Architecture is a compiler assisted architecture. Much of the burden of hardware coordination and synchronization is placed on software. The branch decoupling compiler applies the decoupling algorithm as well as traditional compilation techniques to regular high-level language source files to generate the two-stream binary executables. Besides the hardware architecture, the BDA also has an associated system software architecture and these are described in the next few sections.

2.2 Hardware Architecture

2.2.1 The Branch Decoupled Processor

The Branch Decoupled (BD) Processor consists of two components, the P-processor and the B-processor. Each processor executes instructions that belong to the respective streams. Figure 2.3 shows the architectural block diagram of the BD processor. Each processor core in the BD processor is a general purpose superscalar processor.

Each processor has its own set of program counter (PC) register, instruction fetch, decode, issue logic and functional units. Instruction fetching, decoding, issue and execution of each processor are independent from those of the other processor. This effectively doubles the issue and commit bandwidth of the BD processor. The two processors maintain identical control-flow paths with the help of architectural queues between the two processors. Hardware queues are used for inter-processor communication and thus the processors run asynchronously and exploit instruction level parallelism (ILP) within their streams.

The configurations for the BD processor can be chosen from a gamut of design choices. There could be separate instruction caches (I-caches) for each of the processors or both can
share a single unified I-cache. The same applies in the case of data caches (D-cache). There
could be mixed cache configurations too, for example a unified I-cache and separate D-caches
or a unified D-cache and separate I-caches. One deciding factor in choosing a configuration
could be the application architecture used for the BD processor. Since the instruction address
spaces are different for the two processors, separate I-caches can result in higher hit rates. Also,
since the two streams operate on disjoint data sets, separate D-caches might be beneficial. The
BD processor considered in this thesis uses separate I-caches and a unified D-cache.

Both processors also have separate register files and function units. Their configurations
are orthogonal and the two processors might have a different number of registers and function
units. However, having different registers might unnecessarily complicate the task of register
allocation by the compiler. Previous work on the BDA [28] and [17] assumed that the B-
processor did not have the capability of processing floating point operations. However, in this
thesis, the B-processor configuration is chosen to allow it to process floating point operations.
This allows more flexibility in the decoupling algorithm used by the compiler.

The BD processor follows a shared memory model. In spite of being independent, the two processors within the BDA share the same memory address space. However, the data sets on which the processors operate might be disjoint within the shared memory.

2.2.1.1 **Branch Processor**

The primary task of the B-processor is to execute instructions contained in the B-stream and to determine the control-flow paths for both the processors. The B-processor computes the branch conditions and the BTAs for itself. It computes the branch conditions for the program processor and forwards the results through an architectural queue. This allows both processors to have identical control-flow paths during execution and to maintain correct program execution semantics.

The program counter (PC) of the B-processor functions in the traditional way. It is automatically incremented by the size of an instruction and points to the next sequential address or in the case of a taken branch, it is set to the BTA. So, with respect to handling branches, the B-processor still suffers all the branch penalties of program execution. Branch penalty reduction techniques like dynamic branch prediction, speculative execution can be employed to reduce branch related stalls in the B-stream.

The instruction set of the B-processor consists of control-flow instructions, integer and floating-point computation instructions and instructions which access memory. In addition, the instruction set is augmented with new copy and synchronization instructions special to the branch decoupled architecture paradigm. The branch instructions on the B-processor insert the computed branch conditions into an architectural queue called the *Branch Condition Queue* (BCQ). These values are dequeued by the P-processor.

2.2.1.2 **Program Processor**

The P-processor executes the instructions in the P-stream by using the control-flow information from the B-processor. The instruction set is identical to that of the B-processor. The
P-processor uses the outcomes of the conditional branch instructions that are computed by
the B-processor to decide the flow of execution. The branch handling mechanism is
explained in more detail in the next section.

2.2.2 Control Communication Mechanism

The control communication mechanism is important for the B-processor to transfer the
computed control-flow information to the P-processor. The communication structure can be
explained by a typical example of decoupling.

Consider the high-level language source code in Figure 2.4. The corresponding assembly
translation is shown in Figure 2.5. Assume the registers $2, $3, $4, $5 hold the values of i,a,b,c
respectively.

```
if ( i >= 0 )
    a = b + c
else
    a = b - c
```

Figure 2.4 Source code to be decoupled

The instruction stream is decoupled into the two streams as shown in Figure 2.5. The
conditional branch instruction, `bltz` is decoupled into the B-stream and a corresponding `bfq`
instruction is placed in the P-stream. The B-processor computes the branch condition and
copies a token indicating whether the branch was taken or not-taken (T/NT) into the BCQ.
When the P-processor encounters a `bfq` instruction (branch), it treats it like an unconditional
jump instruction. The BCQ is checked to see if there is an entry. If an entry is found in
the BCQ, this token is copied by the P-processor. When the `bfq` instruction commits, the
P-processor uses the token value to determine whether the PC value needs to changed to the
target address or incremented to the next linear fetch address. If the BCQ is empty, it copies a
request token into the BCQ. When a conditional branch is committed on the B-processor, the
BCQ is checked and any request token is removed and the P-processor is allowed to continue.
In the example, when the branch instruction (bltz) is executed by the B-processor, an entry is added to the BCQ indicating whether the branch was taken (1) or not (0). The bfq, branch from queue instruction, branches to the address contained in the instruction if the corresponding BCQ entry is 1. If not, the next instruction is fetched from the next sequential address.

![Diagram of decoupled streams](image)

Figure 2.5 The decoupled streams

### 2.2.3 Data Communication Mechanism

Although maximum decoupling between the two processors is desirable, there are situations when the data produced by one stream is required by the other and vice versa. In this case, the data needs to be copied between the two streams. Architectural queues provide quick
datapaths for data transfers between the two processors. The queues are also designed to be non-blocking so as to reduce the coupling between the processors and increase asynchronism.

Both processors are capable of processing instructions which read and modify memory. Since the BD processor follows a shared memory model, there are opportunities for memory conflicts if loads and stores on either side alias into the same location. This requires good memory alias analysis to be done by the compiler to identify potential memory conflicts. Synchronization points in the form of sync instructions need to be inserted by the compiler so that the loads and stores on either stream are scheduled correctly according to program semantics. The compiler inserts syncs into the instruction encoding to indicate that the load/store instruction that has a sync_after must be executed before a corresponding load/store instruction that has a sync_before. The BD processor uses sync queues to maintain the right order for memory instructions. For example, consider a load instruction A that reads from location M in the P-stream. The compiler determines that a store instruction B aliases with instruction A. Then the store must be executed before the load to preserve correctness. Thus the compiler encodes a sync_after into the instruction B and a sync_before into the instruction A.

Copies and syncs in the BDA are done implicitly. The instruction is encoded by the compiler to indicate which of the source operands need to be retrieved or copied into one of the copy queues before the instruction is issued. A field in the instruction opcode contains 1 bit flags for each destination as well as source operand. A true corresponding to source operands indicates that the source operand needs to be retrieved from the copy queue instead of the register file. Correspondingly, a true in a destination field of an operand indicates that when the instruction commits, the operand value needs to be copied into the copy queue. Syncs are handled in a similar manner. A 2-bit field is used to indicate syncs. A true in the sync_after flag indicates that a token needs to be placed into the sync_queue. A true in the sync_before field indicates that the instruction can be issued only if a corresponding token can be found in the sync queue that it reads.

Figure 2.6 shows an example of decoupling with implicit copies. The value of $3$ produced by the first instruction in stream-1 is consumed by the second instruction in stream-2. This
value is transferred by an \textit{implicit} copy and is shown in italics. An annotation field in the instructions’ opcodes indicates the necessary copy values. The implementation details of this process by the compiler is explained in more detail in the next chapter.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig2.6.png}
\caption{Decoupling with implicit copies}
\end{figure}

**Dynamic Scheduling with Copies and Syncs** In the case of the BDA, an instruction may have its input source operands that need to be obtained from either the register file or a copy queue. In addition, to having all its input dependences satisfied, an instruction can be issued only if it is \textit{sync\_ready}. A load or store instruction is \textit{sync\_ready} if it has no load or store on the other stream that might lead to a memory conflict.

Dynamic scheduling within the BDA must handle copies and syncs. The commit stage of the pipeline is natural choice for a copy instruction to provide a value to the copy queue. Since the two processors have independent pipeline stages, copy values can be provided during the
commit stage. Consider the scenario where a speculative instruction copies a value into the copy queue and the consumer instruction retrieves this value. If the consumer commits before the speculation on the other processor is discovered to be wrong, then the program semantics would be violated. So the copying values to the copy queue during commit stage guarantees a correct value to the consumer. The process is similar in the case of load/store syncs.

2.2.4 Software Architecture

The difference between the branch decoupled architecture paradigm and traditional paradigms is that the BDA requires extensive software support. A full set of designated system software ranging from operating system, compiler, libraries to binary utilities are required to generate and execute the two-stream decoupled binaries in the BDA. The software architecture of the BDA is shown in Figure 2.7.

2.2.4.1 System Software Architecture

The BDA system software consists of the both the development environment to generate the executable binaries and the execution environment that is needed to load and execute the binaries.

2.2.4.2 Development Environment

An integral component of the development environment is the branch decoupling compiler (BD compiler) [28]. The BD compiler makes the decoupling process transparent to the programmer. It operates on the high-level source files and generates the two-stream executable.

In addition to traditional compiler passes, the BD compiler is augmented with new passes for decoupling. The decoupling pass consists of dependence, alias, data-flow analyses and the actual decoupling algorithm. Dependence and alias analyses construct a graph of data and control dependences in the program and the data-flow analysis is used to propagate dependences beyond basic blocks. The decoupling algorithm identifies instructions to be placed into the respective streams based on the information from the previous passes. The compiler is also
The BD compiler generates an assembly file for each of the high-level source file. The assembler of the BDA transforms each assembly file into an object file and the linker links the object files with decoupled libraries to generate the final executable.

2.2.4.3 Execution Environment

The execution environment is responsible for loading the executables into the main memory by transforming BDA executables from their binary layouts into their memory layouts. After the BDA executable is loaded into the memory, the BDA initializes the stack for the executable by pushing program inputs and environment variables on the stack and starts the execution.

2.2.4.4 Binary Layout

The binary layout for the executables for the BD processor are similar to those of the conventional executables except that there are two sections in the BD binaries. The file header at the beginning of the executable defines a magic number for the executable and stores the number of sections including the size and starting point of each section. Sections are placed together subsequently after the header and each section has a header itself with the necessary information for that section (type of the section and offset of section data).

2.2.4.5 Application Memory Layout

Figure 2.8 shows the application memory layout. The text sections are loaded into the memory at the low end of the memory address space and are followed by the data segment. In BDA, the two streams share the same data set. There is a single shared data segment in the memory layout of a BDA application. The two processors share the same stack space. The stack is initialized at the high end of the memory address space and grows downwards. The global heap is located after the end of the data segment and grows upwards.
Figure 2.7 BDA system software architecture
Figure 2.8 Application memory layout
CHAPTER 3. BRANCH DECOUPLING SIMULATION TOOL SET

An overview of the Branch Decoupling Simulation Tool set is presented in this chapter. The tool set includes an out-of-order execution-driven simulator for the BDA and the necessary system software (a compiler, binary utilities and libraries) to support the BDA target. In the later part of the chapter, the various compiler passes developed including a decoupling algorithm based on graph bi-partitioning is presented.

3.1 Overview

3.1.1 The SimpleScalar Tool Set

The SimpleScalar (SS) Tool Set [1] is a suite of simulation tools developed at the University of Wisconsin, Madison. The tool set consists of both functional and performance simulation of modern RISC microprocessors. The SS technical report [1] contains a detailed description of the usage of the tool set, the SimpleScalar (SS) architecture and simulator internals.

The SS (PISA) architecture is a derivative of the MIPS architecture [20]. The tool set consists of five simulators which are capable of fast, flexible, detailed and accurate simulation of the SS architecture. The simulators range from an extremely fast functional simulator to a highly detailed, out-of-order issue, pipelined, superscalar processor simulator that supports non-blocking caches and speculative execution.

The SS tool set also consists a set of application development tools that are targeted towards the SS architecture target. These include a retargeted GNU C compiler, GNU binary utilities (binutils) and GNU C library (glibc).
3.1.1.1 A Brief Overview of the SS Architecture

The SimpleScalar PISA is a derivative of the MIPS architecture. The SS Architecture is equipped with a 64-bit PISA instruction set [1].

Figure 3.1 shows the three different ways in which the instructions are encoded: register, immediate and jump formats.

The register format is used for computation instructions and the immediate format specifies a 16-bit immediate value. The jump instruction format includes a 24-bit jump target. Each instruction format contains a 16-bit annotation field, that can be modified post-compile, with annotations to instructions in the assembly file. This helps to synthesize new instructions or encode information along with an instruction without the need to modify the assembler.

The SS architecture uses a 31-bit address space, with the virtual memory laid out as shown in Table 3.1.

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>Unused</td>
</tr>
<tr>
<td>0x00400000</td>
<td>Start of text segment</td>
</tr>
<tr>
<td>0x10000000</td>
<td>Start of data segment</td>
</tr>
<tr>
<td>0x7fffffff</td>
<td>Stack base (grows down)</td>
</tr>
</tbody>
</table>

Table 3.1 SS Virtual memory space
The BDA architecture takes advantage of the annotation field within the instruction format to encode information about copies and syncs. This mechanism is explained in the later part of this chapter. The virtual memory space is also modified to introduce two text segments instead of one.

3.1.2 Methods for Performance Evaluation

Functional correctness and performance evaluation of microprocessor designs are usually verified using simulators due to the infeasibility of actual hardware prototyping. Simulators come in two flavors: functional simulators and performance simulators.

*Functional simulators* verify the functional correctness of program execution by simulating the machine at the instruction set (ISA) level. *Performance simulators* verify the design of the microarchitecture and these simulators are not concerned with the semantics of program execution. These simulators mainly measure the timing (cycles) of executing instructions.

Performance simulators themselves can be divided into two types: trace-driven and execution-driven [13]. *Trace-driven* simulators work on pre-generated program traces and determine the number of cycles for executing the instructions in the traces. A trace is a dynamic sequence of instructions and can be generated either by software instrumentation, hardware instrumentation or by using the output of a functional simulator.

*Execution-driven* simulators do not work on traces and are coupled with functional simulators. The functional simulator executes the instruction sequences and passes this information to the execution simulator which maintains the timing information. The execution simulators model the timing of pipeline stages in the microarchitecture. By adding a check-pointing capability to the functional simulator, speculative execution and branch prediction can be easily incorporated into the simulation methodology. Figure 3.2 shows the different types of simulators.
3.1.3 Branch Decoupling Simulation Tool Set

The Branch Decoupling Simulation Tool Set is a complete set of tools which has been developed to evaluate the BDA. The tool set facilitates the evaluation of both the hardware and software architecture components of the BDA. Different decoupling algorithms can be plugged into the compiler to evaluate the effectiveness of the decoupling algorithms. The simulators for the BDA allow evaluation with different architectural configurations of the BDA.

The Branch Decoupling simulation tool set consists of an out-of-order execution simulator for evaluating the BDA. It also consists of a GNU C compiler capable of performing decoupling of programs written in the C language. The tool set contains the GNU binary utilities (binutils) and a GNU C library (glibc) for the BDA, both of which allow decoupled binary executables to be easily created. Figure 3.3 shows an outline of the evaluation methodology.

3.2 An Out-of-order Execution-Driven Simulator

In order to simulate the BDA, a true out-of-order execution simulator *sim-bdorder* has been designed based on *sim-outorder*, an execution-driven simulator that comes as part of the
Figure 3.3  BDA evaluation methodology

### 3.2.1 The Necessity for *sim-bdcorder*

Since *sim-outorder* performs execution of instructions in the dispatch stage rather than in between the issue and writeback stages, it requires the values in the register file to be available to an instruction at dispatch. This creates a few problems for the BDA since the BDA follows a dual-processor model. If an instruction needs to wait on a value that is to be communicated by the other stream (a copy), it will stall dispatch unnecessarily. *sim-outorder* was designed to include the execution of instructions at the dispatch stage as this allows easy simulation of perfect branch prediction along with the other different configurations of the processor.

The dispatch stage in *sim-outorder* uses the values in the register file directly. A speculative register file along with the Register Update Unit (RUU) is also maintained to allow speculative execution of instructions which guarantees that a mis-speculated instruction does not modify the register file and thereby allows recovery. *sim-bdcorder* does actual execution of instructions in the issue stage and works with RUU entries rather than with the register file.

### 3.2.2 Design of *sim-bdcorder*

An out-of-order execution simulator requires that the register file not be modified directly before the commit stage. Reservation entries are created for the registers that need to be written, values are read from the register file into this temporary data structure. The instructions are executed out-of-order and actual values are provided in the writeback stage. The commit stage removes the reservation entries for the registers and commits the actual values to the register file. The data structure for the RUU in *sim-bdcorder* is augmented to contain the actual values of registers that the instruction reads and writes. For memory, the load-store queue (LSQ) entry is modified in a similar manner. Stores do not commit to memory until the
Since the design of sim-outorder requires the values of registers to be known at dispatch, the type of values the instructions reads and writes is also stored in this data structure. A new functionality called ruu.execute is introduced for the execution of instructions. This is incorporated into the ruu.issue stage and is invoked when all the dependences of an instruction are satisfied. Though the results are available immediately, the instruction needs to wait until the writeback stage in order to supply values produced to consuming instructions as in a real superscalar processor.

The timing model of sim-outorder is kept intact in sim-bdcorder. The process of recovery is the same as that in sim-outorder and involves simply invalidating the necessary instructions in the RUU.

Figure 3.4 shows the data structure used for describing a reservation station entry in sim-bdcorder. The arrays ip.values and op.values contain the actual values associated with the instruction. The arrays ip.source and op.source are used to indicate whether the input operands are to be obtained from a copy queue or whether the output operands need to be copied into a queue respectively. The entry branch.mispredicted is used by the B processor to communicate the output of the branch condition to the P processor. sim-outorder uses a machine description file for decoding instructions and this is augmented to include the types of input and output dependences.

3.3 Sim-bdcorder Simulator

The sim-bdcorder simulator supports the complete instruction set available in the SS tool set [1]. In addition, there are some special instructions which include the conditional branch instructions, data copy instructions and hardware synchronization instructions.

3.3.1 The BDA Instruction Set

sim-bdcorder makes use of the annotation field in the SS instruction format. The annotation fields are used to indicate implicit copy and synchronization instructions. Figure 3.5 shows
Figure 3.4 Structure for an ruu entry

```c
struct RUU_station {
    S_INST_TYPE IR;
    enum ss_opcode op;
    SS_ADDR_TYPE PC;
    SS_ADDR_TYPE addr;
    /* instruction status */
    int queued;
    int issued;
    int completed;
    int executed;
    int onames [MAX_ODEPS];
    int idep_ready [MAX_IDEPS];
    int idep_names [MAX_IDEPS];
    SS_WORD_TYPE ip_values [MAX_IDEPS];
    SS_WORD_TYPE op_values [MAX_ODEPS];
    int ip_source [MAX_IDEPS];
    int op_source [MAX_ODEPS];
    int branch_mispredicted;
};
```

the possible encoding formats of an instruction: register, immediate, jump and the branch­from-queue (bfq) format. All instructions are 64-bit as in SimpleScalar. The register format is used for computational instructions and has registers as source and destination operands. The immediate format includes a 16-bit constant and the the jump format supports specification of 24-bit jump targets. The bfq format is an instruction format used for the bfq instruction. The offset that is specified with the instruction indicates the value of the program counter for the P processor depending on the value in the Branch Condition Queue (BCQ).

Each instruction format has a 16-bit fixed-location opcode field and a 16-bit annotation field. Figure 3.5 also shows how the BDA uses the annotation field. Copy instructions are indicated implicitly using annotations rather than as explicit copy instructions. Bit positions 14-15 are used to indicate the stream to which the instruction belongs. An encoding of 0x01 indicates the instruction belongs to the B-stream and an encoding of 0x10 indicates the instruction belongs to the P-stream.

Each instruction is assumed to have upto five input dependences and three output dependences. In case of instructions that use double values as inputs, both the register specifiers
<table>
<thead>
<tr>
<th>Format</th>
<th>16-SS annotate</th>
<th>16-opcode</th>
<th>8-rs</th>
<th>8-rt</th>
<th>8-rd</th>
<th>8-ru/shamt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register / copy format</td>
<td>63</td>
<td>32</td>
<td>31</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Immediate format:</td>
<td>63</td>
<td>32</td>
<td>31</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jump format:</td>
<td>63</td>
<td>32</td>
<td>31</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bfq format</td>
<td>63</td>
<td>32</td>
<td>31</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SS annotation field

<table>
<thead>
<tr>
<th>15</th>
<th>10</th>
<th>7</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
</table>

Figure 3.5  BDA instruction encoding
are counted as dependences. For example, if the instruction is \textit{add.} d r3, r4, r6, then the input dependences are r4, r5, r6, r7 and the output dependences are r2, r3. Each instruction is assumed to have a fixed position in the annotation field. Input dependences are assigned locations 0-4 in the annotation field and the output dependences are assigned locations 5-7. A 0 in the annotation bit position for a particular input dependence indicates that the value is available within the instruction's own stream. A 1 in the bit position for an input dependence indicates that the value needs to be obtained from the other stream, that is from a copy queue. A 1 in the bit position for an output indicates that the output value should be copied to the copy queue after it has been produced by the instruction. The BDA compiler is responsible for encoding this annotation information within the instruction.

The bit positions 8-9 are used to indicate synchronization information. These bits are used only in load and store instructions. In order to ensure memory consistency in the dual processor shared memory model, sync instructions are introduced. Sync instructions occur in pairs. Bit 8 is the \textit{sync-before} field and bit 9 is the \textit{sync-after} field. An instruction with the \textit{sync-after} bit asserted indicates that the instruction must enqueue a token in a \textit{sync queue} when it reaches the commit stage. Similarly, an instruction with the \textit{sync-before} bit asserted cannot issue until it finds a corresponding token for itself in the \textit{sync queue}. Bits 12-15 are unused and may be used to encode other information.

Consider the instruction \textit{add} r2, r3, r4. This instruction has two input dependences r3, r4 which correspond to the bit positions 0 and 1 in the annotation field respectively. The instruction also has one output dependence r2 which corresponds to the bit position 5 in the annotation field. If the value r4 needs to be obtained from the other stream, a 1 is encoded in position 1 of the annotation field. If the value r2 needs to be consumed by an instruction in the other stream, then a 1 is encoded in bit position 5. Assuming the instruction belongs to the P-stream, the annotation field for the instruction field is encoded as 10100010.
3.3.2 The Program and Branch Processors

The Branch Processor The Branch processor is a superscalar processor that contains both integer and floating point functional units. It has its own branch prediction mechanism and different branch prediction techniques can be applied. All the branch instructions executed in the B processor enter a value in the BCQ (branch condition queue). The branch processor also communicates with the program processor through the sync, p2b and b2p queues. These queues are presented in the next section. The branch processor has 32 general purpose integer registers, 32 floating point registers, 2 result registers and a floating point condition code (FCC) register.

The Program Processor The Program processor is also a superscalar processor. It too has an integer and a floating point register file like the B processor. The branch instructions in the P processor are called bfq or branch from queue instructions. The branch instructions branch to a certain target depending on the value it finds in the branch condition queue. The P processor communicates with the B processor through the queues mentioned above.

Table 3.2 shows the BDA architecture register definitions.

<table>
<thead>
<tr>
<th>Hardware Name</th>
<th>Software Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>$zero</td>
<td>Zero-valued source/sink</td>
</tr>
<tr>
<td>$1</td>
<td>$at</td>
<td>Reserved by assembler</td>
</tr>
<tr>
<td>$2–$3</td>
<td>$v0–$v1</td>
<td>Function results return registers</td>
</tr>
<tr>
<td>$4–$7</td>
<td>$a0–$a3</td>
<td>Function argument registers</td>
</tr>
<tr>
<td>$8–$15</td>
<td>$t0–$t7</td>
<td>Caller saved registers</td>
</tr>
<tr>
<td>$16–$23</td>
<td>$s0–$s7</td>
<td>Callee saved registers</td>
</tr>
<tr>
<td>$24–$25</td>
<td>$t8–$t9</td>
<td>Callee saved registers</td>
</tr>
<tr>
<td>$26–$27</td>
<td>$k0–$k1</td>
<td>Reserved by operating system</td>
</tr>
<tr>
<td>$28</td>
<td>$gp</td>
<td>Global pointer</td>
</tr>
<tr>
<td>$29</td>
<td>$sp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>$30</td>
<td>$s8</td>
<td>Callee saved register</td>
</tr>
<tr>
<td>$31</td>
<td>$ra</td>
<td>Return address register</td>
</tr>
<tr>
<td>$hi</td>
<td>$hi</td>
<td>High result register</td>
</tr>
<tr>
<td>$lo</td>
<td>$lo</td>
<td>Low result register</td>
</tr>
<tr>
<td>$fcc</td>
<td>$fcc</td>
<td>Floating point condition code</td>
</tr>
</tbody>
</table>
### 3.3.3 Architectural Queues of the BDA

The BDA has five architectural queues which are used to transfer control and data information between the two processors. The branch condition queue (BCQ) between the branch and program processors conveys branch conditions to the program processor. There are two queues in each direction called `copyq_p2b` and `copyq_b2p` for handling copy instructions. In order to handle synchronization instructions for loads and stores, there is a pair of queues in each direction.

The queues are implemented in the simulator in a similar manner. Each entry in the queue contains a tag to the reservation station entry for the instruction that is waiting (sending) a value from (to) the queue. The queues are implemented as circular queues with head and tail pointers.

#### 3.3.3.1 The Branch Condition Queue

Figure 3.6 shows the structure of the BCQ. Each entry in the queue contains a valid flag, pointers to the reservation station and fetch record entry, a flag to indicate which of the two are valid and flags to indicate whether a value has been entered into the queue or requested from the queue. The queue also contains pointers to the head and tail entries and the number of values entered and requested.

The BCQ operates in the following way. When a request for a bfq entry is received and an entry is present in the BCQ, the entry is read by the requesting instruction and that instruction becomes `branch ready`. If no entry is present in the BCQ, the instruction creates a request entry in the BCQ. If a branch on the B side commits, it supplies a value to the BCQ and the blocked bfq instructions are woken if all their input dependences are satisfied.

#### 3.3.3.2 The Sync Queues

There are two sync queues in the BDA for each processor. The structure of the queues is similar to that of the bcq and is shown in Figure 3.7. The entry in the queue contains a pointer to the reservation station of the instruction. The entry does not contain values.
struct bfq_entry {
    int valid;  /* this bfq entry is valid */
    struct RUU_station *rs;  /* points to ruu_station entry */
    struct fetch_rec *frec;  /* fetch record */
    int rs_valid;  /* indicates which of the above 2 is valid */
    int entered;  /* flags */
    int requested;
};
struct bfq_queue {
    int num_entered;  /* number of entries in the bfq */
    int num_requested;  /* number requested */
    int num_requested_tail;  /* pointer to tail of last requested token */
    int num_entered_tail;
    int head;
    struct bfq_entry entries [MAX_BCQ_LEN];
} bfq;

Figure 3.6 Structure of the branch condition queue

but tokens which indicate precedence among the loads and stores. When a sync.after bit is asserted in an instruction, a token is entered in the sync queue when the instruction commits. An instruction with the sync.before bit asserted attempts to remove a token from the queue. The operation is similar to that of the bcq.

3.3.3.3 The Copy Queues

The BDA is equipped with two copy queues which are responsible for data transfer. The structure and operation of the copy queues is similar to that of the sync queues.

3.4 Branch Decoupling Compiler

The Branch Decoupling Compiler is one of the most important components of the simulation tool set. A compiler that is capable of performing decoupling is essential to the operation of the BDA. The branch decoupling compiler (bdcc) is based on the GNU C Compiler [24] that is included in the SS tool set. It is augmented with control-flow, data-flow, dependence and alias analyses and a decoupling algorithm. The decoupling process in the compiler usually takes place as the last pass before assembly code generation. The compiler uses an interleaved
struct syncq_elem {
    int valid;
    int token_requested;
    int token_entered;
    struct RUU_station *rs;
};

struct syncq {
    int num_requested;
    int num_entered;
    int head;
    int num_entered_tail;
    int num_requested_tail;
    struct syncq_elem entries[MAX_QLEN];
};

struct syncq syncq_p2b;
struct syncq syncq_b2p;

Figure 3.7 Structure of the sync queue

text segment interpretation for the purpose of applying traditional compiler techniques. The final binary executable generated however consists of two text segments for each processor.

### 3.4.1 The GNU C Compiler

This section provides a brief introduction to the compilation process followed in the GNU Compiler Collection (GCC). The GCC is a highly portable and retargetable compiler. The gcc program is actually a driver program which is responsible for initialization, decoding arguments, opening and closing files and sequencing passes. The gcc driver program determines which compiler (C, C++, Java etc.) needs to be invoked depending on the source file extension provided to it.

A parsing pass is first invoked which parses the entire input. This pass converts the input into a tree representation which is then converted to the the RTL IR language using the target machine description file. Each time the parsing pass reads a complete function definition, all other compiler passes (tree and RTL generation, tail call optimization, SSA optimization, local and global CSE, loop optimization, register allocation and scheduling) are run in sequence, ending with the output target machine assembly code.
3.4.2 Register Transfer Language - GCC’s IR

Most of the work in the branch decoupling compiler is done on an intermediate representation called register transfer language (RTL) [24]. In this language, the instructions to be output are described one after another in an algebraic form based on Lisp lists that describes the function of the instruction. Each statement in the program is represented as a syntax tree. The RTL generation pass converts this syntax tree representation into RTL. The RTL code for a function is a chain of RTL nodes. Each node represents an instruction as an\textit{rtx} expression.

The \textit{rtx} expression can represent most of the instruction types (computational instructions, memory operations, jump and call instructions). All the passes of the compiler work using this IR and the assembly is generated by using a machine description file. The machine description file lists templates for each instruction, so each \textit{rtx} expression is looked up in this file and a matching assembly template is selected. Each \textit{rtx} expression may expand into multiple assembly instructions.

In order to understand the RTL, consider a simple instruction: \textit{addu} $r3$, $r1$, $r2$. This can be represented in RTL as (\textit{insn}:\textit{(set}((\textit{reg}: 3} plus ((\textit{reg}: 1} (\textit{reg}: 2)))). The \textit{insn} code is used to represent a simple machine instruction. Different codes are used to represent other types of instructions like jumps and calls. The \textit{set} operation is used to indicate that the result of the \textit{plus} operation on the registers \textit{reg 1} and \textit{reg 2} is to be placed into the register \textit{reg 3}.

3.4.3 Control and Data Flow Analysis

The purpose of control and data flow analysis within branch decoupling is to identify dependences that may exist between instructions that belong to different basic blocks in a program’s flow of execution. This is required so that the necessary copy information can be included along with the instruction.

The control and data flow analysis pass starts with constructing a control flow graph for the function. Control flow analysis depicts the precedence relationship among instructions for correct program execution. Data flow analysis is used in decoupling to propagate dependences between instructions beyond basic blocks.
RTL nodes with data and control dependence relationship are connected directly or indirectly by edges. The resulting control flow graph may be cyclic due to the presence of loops. The algorithms and data structures used to construct data and control flow graphs are derived from well known data-flow techniques [16].

The control-flow graph of a function is constructed in two stages. First, the RTL chain of a function is broken into basic blocks and a directed acyclic graph (DAG) [16] is constructed for each basic block. The DAG depicts the data dependences between instructions within a basic block. Next, the basic blocks are considered as nodes and a control flow graph (CFG) is constructed for the function.

Once the CFG for a function is constructed, data flow analysis is applied to construct the use-definition chains (U-D) for the instructions. A use of a register indicates its use as a source operand. A definition indicates either an assignment to this register or a memory load into the register. The U-D chains are used to propagate dependence information that may exist beyond basic block boundaries.

3.4.3.1 Basic Block Identification

During control flow analysis, the RTL chain for a function is broken into basic blocks. In RTL, every RTL node has one of the following six expression codes:

**INSN** computational instructions that are not jumps or function calls

**JUMP_INSN** direct and indirect jumps, unconditional and conditional branches

**CALL_INSN** function calls

**CODE_LABEL** labels denoting jump instruction targets

**BARRIER** indicates that control flow cannot pass this point

**NOTE** debugging and declarative information

The RTL chain is traversed and basic blocks are identified as starting at

- the first instruction of an RTL chain
• a CODE_LABEL

• an INSN or a NOTE after a JUMP_INSN, a CALL_INSN or a BARRIER

and ending at

• the last instruction of an RTL chain

• an INSN or a NOTE before a CODE_LABEL, a JUMP_INSN, a CALL_INSN or a BARRIER

3.4.3.2 Dependence Analysis and Directed Acyclic Graphs

After the basic blocks are identified, dependence analysis is performed by analyzing the instructions in the RTL chain. This identifies all the true dependences between instructions within a basic block. Dependence analysis also results in the creation of a directed acyclic graph for each basic block. For memory related operations, alias analysis is also applied at this stage. The alias analysis technique used is described in the next section.

3.4.3.3 Control Flow Graph

The nodes of the CFG are the basic blocks that are identified using the technique described above. The CFG describes the control flow relationships among the basic blocks in the function. In constructing the CFG, the successors and predecessors are identified as follows. $B_1$ is a predecessor of $B_2$, and $B_2$ is a successor of $B_1$ if

1. there is a conditional or unconditional jump from the last statement of block $B_1$ to the first statement of $B_2$, or

2. $B_2$ immediately follows $B_1$ in the order of the program and $B_1$ does not end in an unconditional jump.

The CFG of a function is constructed by identifying predecessors and successors of all the basic blocks.
3.4.3.4 Use-Definition Chains

The use-definition chains of the instructions is constructed to identify the dependences that may exist beyond basic block boundaries. This information is essential to identify any copies that may be required between the two streams.

Iterative data flow analysis with backward propagation is used to construct the u-d chains. For every basic block $B$, three sets of RTL nodes are defined for every register $r$:

$\text{GEN}[B][r]$ the set of definitions of register $r$ that are generated within the block $B$ and reach the end of the block.

$\text{IN}[B][r]$ the set of definitions of register $r$ reaching the entry of $B$

$\text{OUT}[B][r]$ the set of definitions of register $r$ reaching the exit of $B$

The relationship among the three sets can be described using the following data-flow equations:

\[
\text{OUT}[B][r] = \begin{cases} 
\text{IN}[B][r] & \text{if } \text{GEN}[B][r] = \emptyset \\
\text{GEN}[B][r] & \text{if } \text{GEN}[B][r] \neq \emptyset 
\end{cases} 
\tag{3.1}
\]

\[
\text{IN}[B][r] = \bigcup \text{OUT}[P][r], \text{ P is a predecessor of B} 
\tag{3.2}
\]

Equation 3.1 indicates that a definition $d$ of register $r$ reaches the end of block $B$ if and only if:

- $d$ reaches $B$, i.e., in $\text{IN}[B][r]$, and $d$ is not killed by $B$, or

- $d$ is generated within $B$, i.e., in $\text{GEN}[B][r]$ and is not subsequently redefine within $B$

Equation 3.2 indicates that a definition $d$ of register $r$ reaches the beginning of block $B$ if and only if it reaches the end of one of the predecessors of $B$. First $\text{GEN}[B]$ for all the basic blocks is computed. Data flow analysis by iterations then proceeds with an initial state of $\text{IN}[B] = \emptyset$ and $\text{OUT}[B] = \emptyset$ for all basic blocks. The iteration terminates when the $\text{IN}[B]$'s and $\text{OUT}[B]$'s converge. Then for every use of register $r$ in basic block $B$, $\text{IN}[B][r]$ is the set of definitions for the use, i.e., it is the set of definitions that can reach the use.
3.4.4 Dependence Analysis using Alias Analysis

Since the BDA follows a shared memory model, synchronization points need to be inserted in the instruction stream to guarantee memory consistency. If unnecessary synchronization points are inserted in the instruction stream, then the performance of the BDA degrades since the processors might waste cycles trying to synchronize with the other. If it can be determined that a loads and stores in different streams do not address the same memory location, then synchronization points can be avoided in the BDA.

At the compilation level, alias analysis is used to determine whether two instructions reference the same memory location or not. In the last stage of branch decoupling, alias analysis is used to identify and insert syncs if necessary. Alias analysis identifies all types of dependences between instructions which access memory (read, true, anti and output dependences).

Base-address based alias analysis [2] is used in the branch decoupling compiler. The following types of conditions are used:

1. The base addresses of read and write RTL expressions are different. If both are symbols, there is no conflict.

2. If one address is a stack reference, there is no conflict. Stack references using different base registers do not alias, a stack reference can not alias a parameter and a stack reference cannot alias a global.

3. Addresses involving the frame pointer cannot conflict with addresses involving static variables.

4. Static variables with different addresses cannot conflict.

3.5 Branch Decoupling Algorithm

Branch decoupling can be viewed as a graph bi-partitioning problem where the graph corresponds to the DAG of a basic block or the DAG corresponding to a particular control-flow path of the program. The objective of decoupling can then be thought of as an attempt
to find a partition of the instruction stream into two streams - the B stream and the P stream with a second important objective of minimizing the communication between the two streams. As described before, this is beneficial as it allows the two streams to execute as independently as possible without one having to stall while waiting for a value from the other.

This section presents a branch decoupling algorithm based on a technique which utilizes a combination of graph bi-partitioning and scheduling. The following subsections provide an introduction to the algorithms and their implementation in the compiler.

3.5.1 Kernighan-Lin Algorithm

The Kernighan-Lin algorithm \cite{12} is a specialized simulated annealing algorithm for solving the graph partitioning problem. Natural local search algorithms for determining a bisection of a graph, start with an initial bisection and exchange pairs of vertices across the cut of the bisection if this improves the cut-size. Simple greedy search algorithms may choose the vertex pair that leads to the largest decrease in cut-size. However, these algorithms may lead to a premature solution since there is a possibility of being trapped in a local minimum.

The Kernighan-Lin (KL) algorithm is an iterative strategy which modifies the search procedure to allow the system to escape the local minimum. The search strategy chooses the vertex pair whose exchange results in the largest decrease or the smallest increase if no decrease is possible. In order to understand the algorithm, some definitions are first introduced.

The algorithm starts with an edge weighted graph

$$G = (V, E, W_E)$$

and an initial partition

$$V = A \cup B, |A| = |B|$$

The KL algorithm attempts to find two partitions, X and Y such that $X \subset A$, $Y \subset B$. The algorithms proceeds by swapping nodes between X and Y with the objective of minimizing the number of external edges connecting the two partitions. Each edge is weighted by a cost and the objective corresponds to minimizing a cost function called the total external cost, or cut
Current Weight, \( W = \sum_{a \in A, b \in B} w(a, b) \),

\[(3.3)\]

where \( w(a, b) \) is the weight of edge \((a, b)\).

If the sets of vertices \( X \) and \( Y \) are swapped, \( A_{\text{new}} = (A - X) \cup Y, \ B_{\text{new}} = (B - Y) \cup X \), then the new cut weight \( W_{\text{new}} \) is given by

\[\text{New Weight, } W_{\text{new}} = \sum_{a \in A_{\text{new}}, b \in B_{\text{new}}} w(a, b),\]

\[(3.4)\]

In order to simplify the measurement of the change in cut weight when nodes are interchanged, \textit{external} and \textit{internal} edge costs are introduced. For every \( a \in A \), the following is maintained

\[E(a) = \text{external cost of } a = \sum_{b \in B} w(a, b)\]
\[I(a) = \text{internal cost of } a = \sum_{\hat{a} \in A, \hat{a} \neq a} w(a, \hat{a})\]

The cost difference is the difference between the external edge costs and internal edge costs,

\[D(a) = E(a) - I(a)\]

Similarly for every \( b \in B \), the same information is maintained

\[E(b) = \text{external cost of } b = \sum_{a \in A} w(a, b)\]
\[I(b) = \text{internal cost of } b = \sum_{\hat{b} \in B, \hat{b} \neq b} w(b, \hat{b}) \text{ and}\]
\[D(b) = E(b) - I(b)\]

If any vertex pair \((a,b)\) is picked from \( A \) and \( B \) respectively and swapped, the reduction in cut weight is called the \textit{Gain}, \( g \). This can be expressed as

\[\text{Gain}(a, b) = I(a) - (E(a) - w(a, b)) + I(b) - (E(b) - w(a, b))\]

\[\text{Gain}(a, b) = D(a) + D(b) - 2w(a, b)\]

After the vertices are swapped, the new \( D \) values are computed by

\[D(x) = D(x) + 2w(x, a) - 2w(x, b), \ x \in A - a\]
\[D(y) = D(y) + 2w(y, b) - 2w(y, a), \ y \in B - b\]

The KL algorithm finds a group of node pairs to swap that increases the gain even though swapping individual node pairs from that group might decrease the gain. Some of the terms outlined above are shown in Figure 3.8.
The algorithm is outlined below

1. Initialize partitions A & B and compute total weight, W.

2. Compute D(x) for all vertices x.


4. While there are unlocked vertices do
   (a) Find the unlocked pair \((a_i, b_i)\) that maximizes \(Gain(a_i, b_i)\).
   (b) Mark \(a_i\) and \(b_i\) (but do not swap).
   (c) Update D(x) for all unlocked vertices x, pretending that \(a_i\&b_i\) have been swapped.
   (d) \(i \leftarrow i + 1\).

5. Pick \(j\) that maximizes \(Gain = \sum_{i=1}^{j} Gain(a_i, b_i)\)

6. If \(Gain > 0\) then update

   \[A = (A - \{a_1, a_j\}) \cup \{b_1, \ldots, b_j\},\]

   \[B = (B - \{b_1, \ldots, b_j\}) \cup \{a_1, \ldots, a_j\},\]

   \(Weight = Weight - Gain\)
7. If $Gain > 0$, go to step 2.

Step 4 is executed $|V|/2$ times during each iteration while step 4(a) requires $O(|V|^2)$ time. If the number of iterations is a fixed constant, the total running time of the KL algorithm is $O(|V|^3)$. In the algorithm, the $Gain(a_i, b_i)$ may be negative and this allows the algorithm to escape some local minima.

Since the KL algorithm is heuristic, only a single iteration may result in a local optimum which may not be the global optimum. The heuristic is repeated starting with the new bisection. The algorithm usually terminates in at most five iterations. Another property of the KL heuristic is that handles only exact bisections of graphs. This restriction is eliminated by adding dummy vertices that are isolated from other vertices before the algorithm is applied. An example of the KL partitioning algorithm as applied to the graph in Figure 3.8 is shown in Figure 3.9.

![Diagram](image.png)

**Figure 3.9** Partitioning using KL algorithm

### 3.5.2 Cost Function and Slack Analysis

Since the KL heuristic requires a graph with edges weighted by a cost value and a good initial partition to ensure early termination, a methodology is needed to assign proper cost values and determine an initial partition before the algorithm is executed. This section presents a brief overview of the methodology that is implemented in the compiler.
The cost matrix for the DAG is computed based on slacks [8] of the instructions and the critical path length of the DAG. The cost of each edge $w(a, b)$ is based on the cost function

$$w(a, b) = \frac{CP_{length}}{\min(\text{Slack}_{op_1}, \text{Slack}_{op_2})}$$

where $CP_{length}$ is the critical path length of the DAG and $op_1$ and $op_2$ are the two input operands of the instruction. $\text{Slack}_{op_1}$ and $\text{Slack}_{op_2}$ are the slacks of the instructions which produce $op_1$ and $op_2$ respectively.

In order to compute the critical path, the DAG is first topologically sorted. This involves performing a depth first search and placing each completely visited vertex at the head of a linked list. Weights are assigned to each edge. These weights are based on the latencies of each instruction in the DAG as read from the target machine description file in the compiler. The weights are negated and the single source shortest path algorithm [6] is applied to generate the critical path length. The instructions present on the critical path are also stored. The single source shortest path algorithm is outlined below: (The algorithm assumes that the first node of the DAG is the source)

1. Topologically sort the vertices of the DAG
2. Initialize
   
   (a) for each vertex $v$ set $d[v] = \infty$ where $d[v]$ is the upper bound on the weight of a shortest path from source $s$ to $v$.
   
   (b) $\text{pred}[v] = \text{NIL}$ where $\text{pred}[v]$ is the predecessor of node $v$.
   
   (c) set $d[s] = 0$
3. for each vertex $v$ in the topologically sorted order
   
   (a) for each vertex $v \in \text{Adj}[u]$
      
      
      ii. $\text{pred}[v] = u$
The nodes in the *pred* array are on the critical path and the length of the critical path can be determined by adding the weights of the edges along this path.

In order to determine the slacks of the instructions, the notion of instruction slacks is adopted from [8]. The slack of each node (instruction) is determined by computing the slack of each edge in the DAG. The slack of each edge \( e = u \rightarrow v \) is the number of cycles that the latency of \( e \) can tolerate without delaying the execution of target node \( v \). This is computed as the difference between the arrival time of the last-arriving edge sinking on \( v \) and the arrival time of \( e \). The slack of a node \( v \) is the smallest local slack of all the outgoing edges of \( v \). An example is shown in Figure 3.10.

![Diagram](attachment:image.png)

Figure 3.10 Example of instruction slacks

### 3.5.3 Scheduling

The DAG prescribes the dependences between the instructions, but scheduling needs to be applied to determine the start times of each instruction. After the KL-heuristic is applied, scheduling is performed on both the streams. In certain cases, the instructions in one stream may not be dependent on each other but may be dependent on instructions in the other stream. This happens as decoupling using KL heuristic attempts to swap every vertex pair to determine an optimal partition. As a result, scheduling becomes necessary to determine the order in which the instructions in each stream must be executed.
The scheduling algorithm used in branch decoupling is the simple ASAP (As Soon As Possible) unconstrained scheduling algorithm [15]. This algorithm proceeds by topologically sorting the nodes of the graph. Scheduling of instructions \( \{v_0, v_1, \ldots, v_n\} \) in the sorted order starts with the instruction \( v_0 \). Then an instruction whose predecessors are already scheduled is next selected to be scheduled. This procedure is repeated until every instruction is scheduled. The algorithm is ASAP in the sense that the start time for each operation is the least one allowed by the dependences.

Let \( G_t(V, E) \) be the DAG to be scheduled. The algorithm is presented below

1. Schedule \( v_0 \) by setting time \( t_0 = 1 \).

2. repeat
   
   (a) Select a vertex \( v_i \) whose predecessors are already scheduled.
   
   (b) Schedule \( v_i \) and set \( t_i^S = \max_{j:(v_j,v_i) \in E}(t_j^S + d_j) \)

3. until \( v_n \) is scheduled.

3.5.4 Decoupling Process

This section of the chapter provides an outline of the decoupling pass that is implemented in the compiler. The decoupling pass is applied as the last pass of compilation, before the assembly is generated for the target machine architecture. The decoupling pass is applied on the program, one function at a time and the assembly for the function using the RTL to assembler generation capability in the compiler.

The RTL to assembler code generator in GCC is augmented with an annotation generator which annotates each RTL instruction with the necessary implicit copies and synchronization points as it is converted to assembler output. The annotation generator is explained further in the next section.

Decoupling is performed in three phases. In the first phase, the basic blocks are determined, DAGs and the control-flow graph are constructed. In the second phase, decoupling based on interleaved KL heuristic and scheduling is applied. In the last phase, data-flow analysis is
performed to propagate dependences and the necessary copy instructions and synchronization points are inserted into the instruction stream.

The decoupling pass as implemented by the compiler is outlined below. Each step of the algorithm is implemented as a function in the compiler which performs the necessary task. The function `kl_decouple` performs the decoupling process. It is implemented as an iterative procedure which terminates with the partition for the basic block. The procedure `init_cost_matrix` initializes the cost matrix for the basic block. It determines the edge weights for each edge in the DAG as described in the Section 3.5.2. The procedure `init_kl` uses the critical path information computed by `init_cost_matrix` and initializes the two partitions of the basic block for the actual decoupling process. The heuristic used to determine the initial partitions is explained in the next paragraph. The cost matrix of each DAG is determined using the methodology described in Section 3.5.2 before branch decoupling by the KL algorithm is applied by the compiler.

The instructions (and the terminating branch instruction) that are on the critical path in the DAG are assigned to the *B stream* (*partition A*). All the remaining instructions are assigned to the *P stream* (*partition B*). This ensures that branch conditions are given a priority to be evaluated early by the B stream. This forms the initial partition to the decoupling algorithm. All the instructions on the critical path are assigned a cost of value equal to the critical path length. The remaining instructions are assigned cost weights according to Equation 3.5. The reasoning behind the cost function in Equation 3.5 is that instructions on the critical path (including the terminating branch instruction) must be placed in the B stream as explained above. Thus higher cost weights are assigned to the outgoing edges of instructions on the critical path and lower weights according to Equation 3.5 are assigned to the outgoing edges of the remaining instructions. This ensures that the KL heuristic would attempt to place as many critical instructions as possible in the B stream itself.

In case the size of the partitions are unequal, dummy nodes (which are not connected to any other nodes) are placed in the partitions to make the number of nodes equal in both the partitions. This forms the initial partition to the decoupling process during every iteration.
The \textit{init.kl} also applies the decoupling process based on the KL-algorithm as described in Section 3.5.1 to obtain a partition of the instruction stream. The \textit{schedule} procedure applies the ASAP scheduling algorithm to the two streams. At the end of the current iteration, if there has been no change in the partitions, the decoupling process is terminated. If not, then the process is repeated.

Once the decoupling process is completed, the \textit{propagate} procedure applies data-flow analysis using backward propagation to compute the Use-Definition (U-D) chains for each instruction. This is used to compute dependences that may exist beyond basic blocks since the initial dependence analysis computes dependences only within a basic block. Using this information, the \textit{decouple.2} procedure inserts the necessary copy information into the RTL structure for each instruction. This information is used by the annotation generator to insert the require implicit copy information into the instructions' annotation field. This process is described in the next section. The procedure \textit{final.decoupling} applies alias analysis to determine the required synchronization points into the B and P streams.

The algorithm is outlined below and is also shown in Figure 3.11.

1. \textit{find.basic.blocks} : Analyze RTL chain of the function and identify basic blocks.

2. \textit{construct.dags} : For each basic block, using dependence analysis, construct DAGs.

3. \textit{construct.flowgraph} : Using the RTL chain of the function and basic block information, identify successors and predecessors of each basic block to generate the control flow graph.

4. \textit{kl.decouple} : For each basic block

   (a) $\textit{change} = 0$

   (b) \textit{init.cost.matrix} : Determine critical path of the basic block using single source shortest path. Compute slacks of each instruction and initialize cost matrix.

   (c) \textit{init.kl} : Initialize partitions $A$ and $B$ for the two streams using critical path information according to the heuristic and insert dummy nodes if necessary. Apply KL algorithm to the DAG to obtain a partition.
(d) *schedule*: Apply scheduling to the basic block.

(e) If change in partitions, \( change = 1 \).

(f) If \( change = 1 \) goto step a.

(g) Label the RTL instructions as belonging to either P stream or B stream according to the partitions obtained.

5. *propagate*: Apply data-flow analysis to propagate dependences beyond basic blocks.

6. *decouple_

7. *final decoupling*: Apply dependence analysis using memory alias analysis to determine synchronization points in the instruction stream. Add this to the RTL structure for each instruction.

### 3.5.5 Generating Annotations

The BDA takes advantage of the annotations feature available in the SS tool set. Using annotations, the need for explicit *copy* and *sync* instructions is eliminated. The opcode for each instruction is encoded with the necessary copy and sync information.

In GCC, each RTL instruction is converted into the target machine assembly by using an assembly code generator. The assembly generator uses a target machine description file, which specifies *templates* for each RTL instruction. A template is a single assembly instruction or a set of assembly instructions that accomplishes the instruction’s functionality. The code generator analyzes the RTL instructions and then selects a corresponding template for the instruction from the machine description file.

In order to generate annotations in the assembly code, the code generator in GCC is augmented to support annotations. During decoupling, the copy and sync information is stored in the RTL structure for each instruction. After a template is selected by the code generator, the template is analyzed and the copy and sync information are added to each assembly
Figure 3.11 Decoupling pass design and decoupling algorithm
instruction within the template. The SimpleScalar assembler encodes the annotations in an assembly instruction into a binary format as the executable is generated.

3.6 Branch Decoupled Glibc and Binutils

3.6.1 Branch Decoupled glibc

Branch Decoupled glibc (bdglibc) are C libraries that are targeted to the BDA environment. It is based on the GNU glibc [3][4][7][19][18] that is provided with the SS tool set. The libraries are decoupled so that the binary executables generated by the compiler work with bdglibc.

BDA Binary Start-up Header The BDA binary start-up header is a precompiled BDA object file that is provided with bdglibc. The linker links this header and all the object files together to form the BDA binary executable in the final stage of linking. The binary start-up header consists of an entry point and an exit point that can be understood by the binary loader of the BDA simulator. The simulator initializes the execution environment for the binary executable and transfers control to the entry point in the binary. The program body is then invoked by the entry point and the control is transferred to the exit point upon completion.

3.6.2 Branch Decoupling Binutils

Branch Decoupling binutils (bdbinutils) is a set of branch decoupling binary utilities present in the BDA software environment. The libraries are decoupled so as to work with the BDA toolchain. The assembler bdas and the linker bdld are the major components of the binary utilities. bdbinutils is based on the GNU binutils included in the SS tool set. bdbinutils also includes a utility that has been developed as part of this thesis bds-disasm. This utility generates a two text segment binary executable for the BDA sim-bdcorder simulator.
CHAPTER 4. SIMULATION AND RESULTS

An overview of the simulation methodology is presented first in this chapter. Some simulation results and their analysis are presented next in the remaining part of the chapter.

4.1 Methodology

The simulation environment of the branch decoupled architecture consists of the decoupling compiler, the associated libraries and binary utilities for generated decoupled binary executables. The decoupled binary executables are simulated by an out-of-order execution-driven superscalar simulator which was presented in Section 3.2.

A set of integer and floating point benchmark programs are adopted from the SPEC CPU2000 benchmark suite [25]. These benchmarks are discussed in the next section. The benchmark programs are first compiled with the branch decoupling compiler (with optimization level O3) presented in Section 3.4. The decoupling compiler applies the branch decoupling algorithm on the benchmark program and uses the retargeted toolchain to produce decoupled binary executables. The resulting binaries are then simulated by the out-of-order execution BDA simulator.

4.2 Benchmarks

A combination of both integer and floating-point benchmarks are adopted from the SPEC CPU2000 benchmark suite. The benchmarks used and their input sets are outlined in Table 4.1. During the simulation runs, execution of each benchmark is forwarded by a particular amount of instructions and then execution is simulated for 500 million instructions. The number of
instructions executed and the number of instructions skipped for each benchmark are based on [21]. These are outlined in Table 4.2.

Table 4.1 Benchmarks and input data sets

<table>
<thead>
<tr>
<th>Category</th>
<th>Benchmark</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECfp2000</td>
<td>mesa</td>
<td>-frames 1 -meshfile mesa.in -ppmfile mesa.ppm (test input)</td>
</tr>
<tr>
<td>SPECfp2000</td>
<td>art</td>
<td>-scanfile c756hel.in -trainfile1 a10.img -stride 2 -startx 134 -starty 220 -endx 139 -endy 225 -objects 1 (test input)</td>
</tr>
<tr>
<td>SPECfp2000</td>
<td>equake</td>
<td>&lt;inp.in (test input)</td>
</tr>
<tr>
<td>SPECInt2000</td>
<td>bzip2</td>
<td>input.random 1 (test input)</td>
</tr>
<tr>
<td>SPECInt2000</td>
<td>gcc</td>
<td>cccp.i (test input)</td>
</tr>
<tr>
<td>SPECInt2000</td>
<td>gzip</td>
<td>input.compressed 1 (test input)</td>
</tr>
<tr>
<td>SPECInt2000</td>
<td>mcf</td>
<td>inp.in (test.in)</td>
</tr>
<tr>
<td>SPECInt2000</td>
<td>parser</td>
<td>test.in (test input)</td>
</tr>
<tr>
<td>SPECInt2000</td>
<td>twolf</td>
<td>test (test input)</td>
</tr>
<tr>
<td>SPECInt2000</td>
<td>vpr</td>
<td>net.in arch.in place.in route.out -nodisp -place_only -init.t 5 -exit.t 0.005 -alpha.t 0.9412 -inner_num 2</td>
</tr>
</tbody>
</table>

4.3 Processor Configurations

`sim-bdcorder` which is explained is Section 3.2 is used to simulate the BDA. The system model on which each of the processors within `sim-bdcorder` is based on a typical out-of-order superscalar processor. Table 4.3 contains a description of the baseline architectural parameters.

4.4 Results and Analysis

This section presents the performance of the decoupling algorithm. The results are presented in two parts.

Section 4.4.1 presents the various instruction counts as obtained through the simulations of the BDA. This includes the percentage of instructions that are allocated to each stream by
Table 4.2  Number of instructions executed and amount skipped

<table>
<thead>
<tr>
<th>Program</th>
<th>#Executed (millions)</th>
<th>#Skipped (millions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mesa</td>
<td>500</td>
<td>300</td>
</tr>
<tr>
<td>art</td>
<td>500</td>
<td>1500</td>
</tr>
<tr>
<td>equake</td>
<td>500</td>
<td>1500</td>
</tr>
<tr>
<td>bzip2</td>
<td>500</td>
<td>200</td>
</tr>
<tr>
<td>gcc</td>
<td>500</td>
<td>1000</td>
</tr>
<tr>
<td>gzip</td>
<td>500</td>
<td>40</td>
</tr>
<tr>
<td>mcf</td>
<td>500</td>
<td>1500</td>
</tr>
<tr>
<td>parser</td>
<td>500</td>
<td>250</td>
</tr>
<tr>
<td>twolf</td>
<td>500</td>
<td>400</td>
</tr>
<tr>
<td>vpr</td>
<td>500</td>
<td>190</td>
</tr>
</tbody>
</table>

Table 4.3  Architectural parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue</td>
<td>4-way Out-of-order</td>
</tr>
<tr>
<td>Fetch Queue Size</td>
<td>32 instructions</td>
</tr>
<tr>
<td>Branch Prediction</td>
<td>2K entry bimodal</td>
</tr>
<tr>
<td>Branch mis-prediction latency</td>
<td>3 cycles</td>
</tr>
<tr>
<td>Instruction Queue Size (RUU)</td>
<td>128 instructions</td>
</tr>
<tr>
<td>Load/Store Queue Size (LSQ)</td>
<td>8 instructions</td>
</tr>
<tr>
<td>Integer Functional Units</td>
<td>4 ALUs, 1 Mult./Div.</td>
</tr>
<tr>
<td>Floating Point Functional Units</td>
<td>4 ALUs, 1 Mult./Div.</td>
</tr>
<tr>
<td>L1 D- and I-cache</td>
<td>Each: 128Kb, 4-way</td>
</tr>
<tr>
<td>Combined L2 cache</td>
<td>1Mb, 4-way</td>
</tr>
<tr>
<td>L2 cache hit latency</td>
<td>20 cycles</td>
</tr>
<tr>
<td>Main memory hit time</td>
<td>100 cycles</td>
</tr>
</tbody>
</table>
the compiler. It also includes a count of copies and synchronization points inserted by the compiler. Section 4.4.2 then provides an overview of the performance of the BDA.

4.4.1 Instruction Statistics

This section presents an overview of the instruction statistics obtained by the application of the decoupling algorithm by the compiler. The binary executables of the benchmarks presented in Table 4.1 are simulated to obtain some characteristic information.

Table 4.4 shows the distribution of dynamic branch instruction counts. The table shows the total number of instructions executed and the percentage of branch instructions - conditional, unconditional jumps and call instructions. The table also shows the number of memory references in the execution run of the benchmarks. The distribution of conditional branch instructions indicates the potential benefit of the BDA as these are the instructions that are evaluated by the B processor for the P processor. The distribution of memory references indicates the possible synchronization points between the two processors.

The dynamic instruction counts on both the B and P processors are shown in Table 4.5. This indicates the effectiveness of the decoupling as performed by the compiler. Initially it may appear that the number of instructions on the B side should be less than that on the P side [26]. This statement is true only if there is complete decoupling between the two streams. In the absence of copies between the two streams, a low instruction count on the B side will cause the branch related instructions to be evaluated faster. Thus branch targets pre-computed by the B processor can be supplied to the P processor. However, in the presence of copies, this may not necessarily be true as cycles may be wasted while an instruction on the B processor waits for a copy value to be produced by the P processor. Thus it might be beneficial for attaining a balance between the B and P processors so that there are processor stalls while an instruction waits on a copy.

Figure 4.1 shows the ratio of the instructions in the two streams. Table 4.5 also indicates a column for undecoupled instructions. The glibc that is retargeted to the BDA is not completely decoupled. Thus some function calls may result in the execution of undecoupled instruction
## Table 4.4  Branch instruction and memory reference counts

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total (million) (% Inst Count)</th>
<th>Conditional (% of Branch)</th>
<th>Unconditional (% of Branch)</th>
<th>Calls (% of Branch)</th>
<th>Memory References (% of Inst Count)</th>
</tr>
</thead>
<tbody>
<tr>
<td>art</td>
<td>89.01 (17.8%)</td>
<td>85.10 (95.6%)</td>
<td>2.6 (2.93%)</td>
<td>1.29 (1.45%)</td>
<td>134 (27%)</td>
</tr>
<tr>
<td>bzip2</td>
<td>84.95 (16.9%)</td>
<td>80.81 (95.12%)</td>
<td>3.1 (3.65%)</td>
<td>1.04 (1.23%)</td>
<td>162 (32%)</td>
</tr>
<tr>
<td>gcc</td>
<td>114.36 (22.87%)</td>
<td>89.28 (78.1%)</td>
<td>19.07 (16.67%)</td>
<td>6 (5.24%)</td>
<td>158 (31%)</td>
</tr>
<tr>
<td>gzip</td>
<td>105.63 (21.12%)</td>
<td>80 (75.67%)</td>
<td>19.9 (18.83%)</td>
<td>5.8 (5.48%)</td>
<td>151.88 (30.3%)</td>
</tr>
<tr>
<td>mcf</td>
<td>59.9 (27.71%)</td>
<td>42.28 (70.49%)</td>
<td>11.35 (18.92%)</td>
<td>5.25 (8.75%)</td>
<td>77.90 (36%)</td>
</tr>
<tr>
<td>mesa</td>
<td>125.58 (25.11%)</td>
<td>92.15 (73.37%)</td>
<td>26.95 (21.49%)</td>
<td>6.5 (5.16%)</td>
<td>130 (26%)</td>
</tr>
<tr>
<td>parser</td>
<td>73.34 (25.55%)</td>
<td>40.63 (55.39%)</td>
<td>22.68 (30.93%)</td>
<td>10.02 (13.67%)</td>
<td>93.4 (32.5%)</td>
</tr>
<tr>
<td>twolf</td>
<td>52.80 (22.75%)</td>
<td>43.8 (82.9%)</td>
<td>6.9 (13.16%)</td>
<td>2 (3.85%)</td>
<td>81.48 (35.1%)</td>
</tr>
<tr>
<td>vpr</td>
<td>99.38 (19.87%)</td>
<td>79.53 (80.02%)</td>
<td>15.17 (15.26%)</td>
<td>4.6 (4.7%)</td>
<td>152 (30.4%)</td>
</tr>
<tr>
<td>equake</td>
<td>232.27 (16.87%)</td>
<td>142.67 (61.4%)</td>
<td>63.2 (27.21%)</td>
<td>26.40 (11.37%)</td>
<td>433 (31%)</td>
</tr>
</tbody>
</table>
streams. System calls during the simulation run of the program also result in undecoupled instruction streams. These are executed on the P processor only and the results are copied to the B processor's register file at the end of the instruction stream's execution run. Only the benchmarks mesa, parser and equake seem to have a large percentage of undecoupled instructions. These can be reduced considerably when the libraries for the BDA are completely decoupled. The results for the other benchmarks indicate a reasonable balance of instructions for both the B and P processors. On the average, 48.58% are assigned to the B processor and 38.07% are assigned to the P processor with 13.35% of the instructions being executed in undecoupled mode.

Another metric for evaluating the effectiveness of the decoupling algorithm is the dynamic count of the number of copy and sync instructions that are inserted by the compiler. Table 4.6 shows the distribution of copies and syncs inserted by the compiler. The syncs are inserted after alias analysis is performed by the compiler. The copies indicate the communication required between the two streams. It also indicates the effectiveness of the decoupling algorithm used by the compiler. The benchmark mesa again shows a very small number of these instructions as the number of undecoupled instructions are quite considerable. It should be noted that
since copies and syncs are encoded implicitly into the instructions, these add no overhead to the dynamic instruction count of the execution run.

4.4.2 Performance Statistics

In order to measure the performance of the BDA, the benchmarks were compiled with the GCC compiler that is available in the SS tool set and were simulated using the sim-outorder simulator. The resulting IPC (Instructions per Cycle) for both the system models are shown in Figure 4.2. Figure 4.3 shows the percentage of the time a branch condition is available in the branch queue for the P processor. This gives an indication of the percentage of the time the P processor stalls while waiting for the branch outcome to be computed by the B processor.

![IPC Comparison](image)

*Figure 4.2 IPC comparison between the base and BDA processors*

twolf and parser show the best speedups. The branch conditions are available for both these benchmarks about 90% of the time. The performance of both benchmarks improve by 17.5% and 15% respectively. However, gcc and mcf show performance degradation by 6% and 2% respectively. It can be seen from Figure 4.3 that the branch outcomes are available only 65% and 67% of the time respectively which explains the degradation in performance. The performance improvements for integer benchmarks is about 7.7% on the average. The BDA
Table 4.5 Distribution of instruction counts on each processor

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Instructions Executed (mil)</th>
<th>Instructions on B (mil)</th>
<th>Instructions on P (mil)</th>
<th>Undecoupled Instructions (mil)</th>
</tr>
</thead>
<tbody>
<tr>
<td>art</td>
<td>500</td>
<td>292.8 (58.5%)</td>
<td>209.7 (41.9%)</td>
<td>0</td>
</tr>
<tr>
<td>bzip2</td>
<td>500</td>
<td>269.28 (53.8%)</td>
<td>233.8 (46.7%)</td>
<td>0</td>
</tr>
<tr>
<td>gcc</td>
<td>500</td>
<td>272.19 (54.4%)</td>
<td>218.72 (43.7%)</td>
<td>27.4 (5.5%)</td>
</tr>
<tr>
<td>gzip</td>
<td>500</td>
<td>273.9 (54.8%)</td>
<td>238.4 (47.6%)</td>
<td>7.48 (1.5%)</td>
</tr>
<tr>
<td>mcf</td>
<td>216.45</td>
<td>113.76 (52.5%)</td>
<td>71.46 (33%)</td>
<td>42.28 (19.5%)</td>
</tr>
<tr>
<td>mesa</td>
<td>500</td>
<td>8.3 (1.7%)</td>
<td>3.1 (0.62%)</td>
<td>488.9 (97%)</td>
</tr>
<tr>
<td>parser</td>
<td>287.04</td>
<td>62.79 (21.8%)</td>
<td>64.9 (22.6%)</td>
<td>168.8 (58.8%)</td>
</tr>
<tr>
<td>twolf</td>
<td>232.05</td>
<td>133.45 (57.5%)</td>
<td>95.56 (41.1%)</td>
<td>9.5 (4.1%)</td>
</tr>
<tr>
<td>vpr</td>
<td>500</td>
<td>269.49 (53.9%)</td>
<td>234.74 (46.9%)</td>
<td>10.2 (2.04%)</td>
</tr>
<tr>
<td>equake</td>
<td>1376</td>
<td>481 (30%)</td>
<td>288 (20%)</td>
<td>675 (49%)</td>
</tr>
</tbody>
</table>
Table 4.6 Distribution of copy and sync instructions

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Instructions Executed (mil)</th>
<th>Synchronization Points (mil)</th>
<th>Copies (mil)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>cp2b</strong></td>
</tr>
<tr>
<td>art</td>
<td>500</td>
<td>24.25</td>
<td>48.97</td>
</tr>
<tr>
<td>bzip2</td>
<td>500</td>
<td>147.53</td>
<td>82.16</td>
</tr>
<tr>
<td>gzip</td>
<td>500</td>
<td>48.36</td>
<td>52.38</td>
</tr>
<tr>
<td>mcf</td>
<td>216.46</td>
<td>21.79</td>
<td>13.13</td>
</tr>
<tr>
<td>mesa</td>
<td>500</td>
<td>0.82</td>
<td>0.82</td>
</tr>
<tr>
<td>parser</td>
<td>287</td>
<td>5.27</td>
<td>17.41</td>
</tr>
<tr>
<td>twolf</td>
<td>232</td>
<td>25.31</td>
<td>34.88</td>
</tr>
<tr>
<td>vpr</td>
<td>500</td>
<td>41.87</td>
<td>57.03</td>
</tr>
<tr>
<td>equake</td>
<td>1376</td>
<td>160.35</td>
<td>73.86</td>
</tr>
</tbody>
</table>
performs well on floating point benchmarks with about 5% performance improvement on the average.

![Availability of branch conditions](image)

Figure 4.3 Availability of branch conditions when required
CHAPTER 5. CONCLUSION AND FUTURE WORK

This chapter concludes the thesis and provides an insight into the future work in this direction.

5.1 Conclusion

This thesis presented three facets of a simulation tool set developed for the Branch Decoupled Architecture paradigm. First a retargetable toolchain consisting of a branch decoupling compiler, binary utilities (assembler, linker) and libraries for the BDA platform was introduced. Next a decoupling algorithm based on graph bi-partitioning and scheduling used by the compiler was introduced. Finally, an out-of-order execution driven simulator for simulating the performance of the BDA was presented.

The decoupling algorithm used by the compiler achieves a good balance between the instruction streams executed by the B and P processors. On the average, about 48.6% of the instructions are executed by the B processor and 38.1% of the instructions are executed by the P processor. The remaining 13.3% of the instructions are undecoupled instructions that need to be executed only the P processor.

The number of synchronization points introduced by the decoupling algorithm are also much lesser than that in [28]. Since the copies and syncs are done implicitly by the compiler requiring no explicit instructions, the dynamic instruction counts are not increased. The BDA shows performance improvements over the base system model, with about 7.7% and 5.5% performance increase in case of integer and floating point benchmarks respectively.
5.2 Future Work

This section enumerates some directions of future work.

1. The decoupling algorithm implemented by the compiler performs the decoupling process on basic blocks. Using feedback profiling techniques, the actual computation backbone of the program can be identified. The backbone consists of the most frequent control-flow paths in the program. The decoupling can be more effective if it is applied on these identified paths as a whole.

2. Different variation of the cost function used to determining cost weights for the edges in the DAG can lead to different decoupling results. By applying more interesting cost functions, better decoupling results could be obtained.

3. From previous work on trace-based decoupling, it was found that a combination of branch prediction and decoupling allows the BDA to perform well over the base system model. This technique can be incorporated in the execution-driven simulator to study the performance impact of this technique.

4. The compiler has been implemented in such a way that different decoupling algorithms can be plugged into it. Future work might involve developing different decoupling algorithms and analyzing their behavior.

5. A further research direction could be in whether the BDA can be implemented as an SMT architecture where the P-stream and B-stream execution could share hardware resources.
Bibliography


