Schedule Construction Using an Expert System

Joshua Scoggins

March 16, 2012

Contents

1 Problem Domain Description and Justification

Most modern high-performance processors take advantage of a technique called Instruction Level Parallelism (ILP) to potentially reduce the running time of a given sequential program by simultaneously executing several instructions per clock cycle [1]. This potential is tempered by the following factors:

1. The amount of parallelism in a given program
2. The parallel capabilities of the processor
3. The ability to identify parallelism from a sequential program.
4. The ability to construct the best parallel schedule possible based on system constraints

The amount of parallelism in a given program is directly affected by the amount of dependent instructions within the program. A program with a high frequency of data dependent instructions will not gain nearly as much of a performance improvement as a program that performs scientific computations where many operations are independent of each other. Even if the processor is capable of executing several instructions per cycle the mix of instructions may still cause instructions that can be executed in parallel to be executed sequentially due to processor limitations.

For example, lets say that a processor can execute an integer and floating point operation per cycle. Sections of integer or floating point exclusive code will still cause sequential execution to occur. Thus for this theoretical processor to execute more than one instruction per cycle a program consisting of an even mix of floating-point and integer instructions required to maximize performance of the given program. This is not only hard to do consistently but may not even be possible in some cases.

When it is possible, identification of parallelism within a sequential program consists of extracting parallelism and scheduling it for parallel execution. Performing this identification is done dynamically in hardware (Superscalar), statically by software (Very Long Instruction Word), or through a combination of hardware and software means (Explicitly Parallel Instruction Computing)[1].

1.1 Superscalar Architectures

Most modern computer processors are superscalar. This means that the hardware performs dependency analysis on the instruction stream at runtime to detect dependencies between a very small set of instructions. Instructions that are found to not have any dependencies are dispatched to an appropriate functional unit by the scheduler. This has quite a large number of benefits. The biggest being that the programmer visible instruction set is separate from the internal instruction set. This is achieved through the use of a code translation layer that translates the exposed instruction set into the internal instruction set of the processor. This abstracts away the need to know memory access times and most implementation specific features about a given architecture. This allows programs compiled for the externally visible architecture to run on all revisions of the architecture without the need for re-compilation.

The biggest problem with superscalar processors is that dynamic nature of the hardware scheduler means that long spans of instructions can't be dynamically scheduled while maintaining sane transistor counts, thermal properties, or cycle times. This limits the identification of parallelism to a small set of instructions. The other problem with superscalar processors is increasing the number of functional units requires the scheduling hardware to be updated to take these new units into account. Too many functional units will prevent the hardware scheduler from taking complete advantage of all the available execution resources due to time constraint the scheduler is under[1]. Thus it is up the hardware designer to balance the number of functional units with the complexity of the associated hardware scheduler.

1.2 Very Long Instruction Word

Very Long Instruction Word (VLIW) exploits ILP by using a purely software approach to parallelism identification and schedule construction. The idea behind VLIW is that the area consumed by a hardware scheduler could be better used by including more functional units and registers. These increases allow the processor to execute more operations per cycle and keep more data locally within the processor which also increases the ability to execute more operations per cycle. The term VLIW refers to the fact that processors of this type operate on groups of instructions bundled into very long instruction words. These instructions encode not only the operation but also which functional unit the instruction should be dispatched to. This change is necessary to ensure that the large number of functional units can be fed with information quickly instead of waiting for enough single instructions to be built up to perform a dispatch. Since VLIW uses static scheduling it is up to the compiler to perform all of the features of a hardware scheduler including stalling. However, this approach does give the scheduler far more time to analyze far larger sections of a program for parallelism. This requires a compiler writer to be intimately familiar with how the architecture works. This includes information such as memory access times, functional unit count, register file layout, and even pipeline latencies.

This kind of knowledge is very implementation specific and changes in the number of functional units, access times, and even register counts across different generations require programs to be re-compiled to ensure system correctness. This has the benefit that improvements in the performance of a compiler will directly improve performance of the overall system. The downside to this is when the compiler's scheduling facilities are less than stellar. Most compiler writers will not totally understand the VLIW philosophy and as such will usually only implement simple scheduling algorithms that are easy to implement and not error-prone. More advanced region scheduling algorithms are either continually relegated to the next version of the compiler or are just not implemented at all. The need to re-compile programs to reflect changes across the generations of a VLIW architecture makes it nearly un-maintainable for a commercial software vendor that supports multiple architectures. Either the vendor has to ship the source code to their program to be compiled by the user on their machine or the vendor must include every possible version with their installation media. This is a deal breaker because neither version will allow vendors to provide a high quality service. These problems have relegated VLIW to areas of embedded systems and high

[1] usually the hardware scheduler has maybe 5 clock cycles to perform analysis in!

performance graphics cards where the need to recompile code to take maximum advantage of the hardware is not only commonplace but required for optimal performance.

1.3 Explicitly Parallel Instruction Computing

Explicitly Parallel Instruction Computing (EPIC) architectures exploit ILP through a hybrid software/hardware approach. It is a hybrid VLIW/Superscalar processor that combines the massive parallel capability of VLIW with the compatibility and abstraction of superscalar. Research into EPIC was started in the late 1980s by HP and Intel trying to find a new way to take advantage of ILP without having to resort to increases in clock speed, pure dynamic scheduling, or improvements in branch predictor accuracy. Performance improvements stemming from these areas were seen to eventually succumb to diminishing returns. The massive parallelism of VLIW was seen as the way forward but the logistical problems of such a design was seen as a very large problem. The research put into EPIC came up with the following tenants:

1. The processor is a target for a compiler
2. The compiler is responsible for extracting ILP out of a sequential program
3. The compiler doesn't need to know the exact number of functional units as dispatch will be handled by the hardware.
4. The processor will operate on fixed long instruction words called bundles
5. Programs compiled for older versions of an EPIC processor will be binary compatible across generations.
6. Memory access latencies are abstracted away by the processor
7. Parallelism is described explicitly
8. The hardware should provide features to assist the compiler in maximizing ILP.

These tenants are quite broad but are a amalgam of features taken from both VLIW and superscalar. It could be inferred that EPIC is really just a VLIW processor with superscalar elements. The problem with this statement is that it isn't very accurate. EPIC is very different from VLIW in many critical areas including: instruction encoding, how parallelism is described, a fully predicated instruction set, the ability to perform speculation, direct control of the branch predictor, and the abstraction of functional unit counts and memory access times. But how does one actually construct a schedule?

1.4 Constructing the Best Possible Schedule

Constructing the most optimal schedule for a basic block is actually NP-complete. This doesn't prevent basic block scheduling from being useful because most basic blocks are quite small and thus it is possible to use a simple scheduling technique known as list scheduling to great effect [1].

Constructing a Data-Dependence Graph

Before we can actually construct the schedule it's necessary to create a data-dependence graph for a given basic block. This graph is denoted G consisting of a set of nodes N where each node is a machine instruction and a set of edges E where each edge represents data dependencies between different instructions[1].

Constructing G consists of the following steps:

1. For each operation n in N create resource reservation table RT_n which contains all of the data read from and written to for operation n. It also encodes what functional units will be reserved by the instruction.

2. For each edge e in E label e with a delay d_e signifying that the destination node must be must be issued no earlier than d_e clocks after the source node is issued.

When dealing with VLIW instructions it is important to note that the second step is extremely critical to ensure proper system performance. On Itanium, and other EPIC based systems, the need to stall waiting for memory loads is handled by the hardware.

These steps will create a data-dependence graph that describes the order of execution and what data will be required at each instruction.

1.5 List Scheduling

List scheduling works by visiting the nodes of the basic block in "prioritized topological order" [1]. It computes the earliest time slot in which each node can be executed based on the constraints imposed by its data requirements and the requirements of the previous nodes. Then the resources needed by the node are checked against a resource-reservation table that collects all the resources committed so far. The node is scheduled in the earliest time slot that has sufficient resources [1]. The algorithm for list scheduling is as follows:

RT = an empty reservation table;
for each n in N in prioritized topological order **do**
  $s = \max_{e=p\to n\in E}(S(p) + d_e)$; {Find the earliest time this instruction could begin, given when its predecessors started.}
  while there exists i such that $RT[s+i] + RT_n[i] > R$ **do**
    $s = s + 1$ {Delay the instruction further until the needed resources are available}
  end while
  $S(n) = s$;
  for all i **do**
    $RT[s+i] = RT[s+i] + RT_n[i]$
  end for
end for

Prioritized Topological Order

It's important to note that list scheduling does not backtrack and schedules each node exactly one time [1]. This requires that nodes be selected on a heuristic priority function based on

which one is "ready" next. There isn't a single heuristic priority function that works for all cases. Instead the priority function must be constructed based on what is the priority for the schedule. Here are some observations about possible prioritized orderings as described in [1]

- Without resource constraints, the shortest schedule is given by the *critical path*, the longest path through the data-dependence graph. A metric useful as a priority function is the *height* of the node, which is the length of a longest path in the graph originating from the node.
- On the other hand, if all operations are independent, then the length of the schedule is constrained by the resources available. The critical resource is the one with the largest ratio of uses to the number of units of that resource available. Operations using more critical resources may be given higher priority
- Finally, we can use the source ordering to break ties between operations; the operation that shows up earlier in the source program should be scheduled first.

Normally, priority function will be the shortest running time or height but when dealing with embedded systems the target will most likely be energy efficiency[4]. In essence, list scheduling is dependent on *what* the compiler writer is wanting to accomplish. This makes list scheduling very flexible but also difficult to grasp if this fact isn't described.

1.6 Justification

The use of a expert system to construct a parallel schedule for a given basic block is perfectly suited to an expert system due to the fact that the knowledge is usually heuristically based. This makes implementing it difficult in a standard programming language without adding pattern matching, automatic data storage, and rule application. These features are inherent in an rule-based expert system like CLIPS and thus makes the act of schedule construction the primary focus instead of the framework the expert system sits on top of.

2 Project Scope

The objective of this expert system is to construct a parallel schedule from a basic block provided by the user. This expert system will operate on Itanium 2 like instructions and generate a list of instructions groups. It is up to the expert system to:

1. Simulate part of an Itanium 2 processor
2. Identify dependencies between instructions
3. Schedule, or rearrange, instructions such that correctness is maintained while maximizing ILP.
4. Not depend on the specifics of the Itanium 2 microarchitecture.

If these conditions are satisfied then the expert system will be considered a success.

3 Knowledge Acquisition

The knowledge required to construct a parallel schedule was primarily acquired through written materials. This included papers on compiler techniques and schedule construction; technical documents on the instruction set architecture of Itanium 2, and books about VLIW and EPIC architectures. These sources were very useful in generating the rules that govern the act of schedule construction and the partial simulation of an Itanium 2 processor.

The only hard part of the knowledge acquisition process has been the acquisition of basic blocks made of Itanium 2 instructions. The problem is that while it is easy to get the material, it is quite difficult to get it formatted in such a way that the schedule constructor will accept it because the act of formatting entails removing nop instructions, removing memory addresses introduced through the use of the objdump command, and finally converting each instruction into a function that CLIPS can directly execute.

This process is quite time consuming and error prone. Thus a C# program was written that converts a file that has been formatted acorrding to the first two formatting rules into the corresponding CLIPS function representation. This had the effect of speeding up the generation of testing data greatly.

The rest of the knowledge that makes up the schedule constructor was acquired through continual iteration during the process of representing and organizing the knowledge acquried.

4 The System

For the sake of brevity, the schedule constructor will now be known as the System. Consequently, the knowledge related to the act of schedule construction will be known as the Runtime. Everything else will be a part of the Architecture.

4.1 The Architecture

The Architecture of the System has four major components: instruction representation, machine simulation, dependency chains, and instruction groups.

4.1.1 Instruction Representation

Since this schedule constructor operates on instructions it is necessary to have a well designed Instruction class that is defined in such a way as to allow it to represent instructions for a wide variety of architectures. The design of the generic Instruction class is a direct copy of the Itanium instruction format which takes the following form

$$(qp)\ op\ \mathrm{reg}_d = \mathrm{reg}_s$$

Where reg_d can be zero, one, or two operands; and reg_s can be zero, one, two , three, or four operands. *qp* is an optional predicate register that allows the instruction to execute

only when the given predicate register is set to one. This is a major feature of Itanium that few other architectures support[2].

However, there are other things that makes up an Itanium instruction. This includes the instruction's execution unit type, and runtime length. Fortunately, there are only six types of instructions in the Itanium 2 architecture. They are floating point (F), extended branch (X), memory (M), integer (I), arithmetic (A), and branch (B) [5]. Each of these instruction types map to one or more *execution units*. These execution units act as a layer of indirection between a subset of the 30 different functional units found in the Itanium 2 backend and the instruction frontend [5]. This has the effect of ensuring binary compatibility across generations of the Itanium architecture where the number of execution units and functional units could change [5]. The use of execution units also simplifies the runtime length of each instruction to four cycles for floating point operations and one cycle for everything else[5].

With all of this information it is now possible to construct a generic instruction object that is defined as having

Name	**Data Type Accepted**	**Description**
GUID	Symbol	Unique ID, usually a gensym
Predicate	Symbol	Predicate Register Name
TimeIndex	Number	Original position in the basic block
ExecutionLength	Number	Cycle count
DestinationRegisters	List Object	List of destination registers
SourceRegisters	List Object	List of source registers
DependencyInformation	DependencyChain Object	Dependency information

Defining an instruction like this has the pleasant side-effect that it can represent an instruction of any arity. It also makes the construction of the object able to be abstracted through the use of functions[3]. This also allows the input to the System to be a series of CLIPS commands.

4.1.2 Machine Simulation

The next component of the Architecture is the machine simulation component which partially simulates an Itanium 2 processor. This partial simulation mainly consists of defining operations and the register file.

Each operation has an associated Operation object and a wrapper function that translates instructions into their associated Instruction object. The Operation object is defined as having an instruction name, the instruction's type, and it's execution length. This information is imbued into an instruction at runtime because the actual operation itself is defined using a wrapper function that takes in a type, the execution length, and a variable list of instructions that are part of this given type. This greatly reduces the amount of work required to define an instruction and ensures that it will work correctly.

The register file simulated is that of an Itanium 2 processor which consists of:

- 128 64-bit General Purpose Registers

[2]The only other architecture that comes to mind is the ARM architecture. But even ARM only supports it in a limited subset.

[3]In fact, there are several default instruction construction functions found in Instruction.clp that do this

- 128 82-bit Floating Point Registers
- 64 One-bit Predicate Registers
- 8 64-bit Branch Registers
- 128 64-bit Application Registers

The definition of these registers takes the form of a series of wrapper functions that take in a numeric range, a prefix, a register class, and the size of the register. These functions construct an instance of each register in the range by concatenating a number in the range with the prefix. This concatenated value is then imbued into the result object with the provided type and storage size. The register file is constructed whenever the System is initialized in response to a basic block being loaded by the user.

4.1.3 Dependency Chains

When defining the Instruction object there was a reference to an object of type DependencyChain. A dependency chain contains all of the dependency information for a given instruction and is quite simple in design. It contains a unique identifier, a list of producers, and a list of consumers. The two lists are meant to describe a producer/consumer relationship between the instruction in question and every other instruction within the current basic block. The idea is that each instruction produces something that is consumed by another instruction at some time in the future. This simple concept is all that is really necessary to construct a parallel schedule.

4.1.4 Instruction Groups

The final major component of the Architecture comes in the form of the InstructionGroup object. An InstructionGroup consists of a list of instructions, a time index, and a boolean value signifying if the InstructionGroup has been printed[4]. Like most other structures, the instruction group comes from Itanium 2 and requires a bit of explanation.

Instructions on Itanium 2 are encoded into 16-byte bundles which consists of three 41-bit instructions and a five bit type code. These bundles are read by the processor's front end which decomposes them into single instructions and dispatches them to the appropriate execution unit. If it found that an instruction can not be dispatched during the current cycle then it is stalled until it is able to be dispatched [5]. The instruction group is a logical extension of the bundle and is defined as being a group of instructions that have no WAW or RAW dependencies[5]. Each instruction group is terminated with an explicit stop which signifies a single cycle stall that allow values currently being computed to progress to a point where the next instruction group can use them. An explicit stop is represented by double semi-colons (;;)[5].

The final type of data hazard, WAR, is handled internally by the Itanium 2 hardware through the use of a scoreboard[5]. This scoreboard will stall groups of instructions that can not be fulfilled due to a level 1 cache miss or because a register is already claimed by another

[4]This ensures that the instruction group is only printed once.

stalled instruction within the scoreboard. This is a viable strategy because the Itanium 2 processor is an in-order design and as such the order of dispatch will automatically be maintained[5].

The instruction groups are also very important for ensuring that the Architecture remains relatively processor agnostic. The explicit stops are generated during the act of printing and are not actually stored within the InstructionGroup object itself. Thus it is useful on a wide variety of architectures for displaying what can be executed in parallel.

4.2 The Runtime

The Runtime consists of four separate phases: imbue, analysis, collect, and schedule. Each of these four phases have a separate purpose that is critical to the proper execution of the System.

4.2.1 Imbue Phase

The Imbue phase is the first phase that occurs after the user loads all of the instructions into the System. In this phase, each instruction is matched to the appropriate operation object defined by the machine. These matches cause the instruction execution length and type to be set if it is found that the instruction is missing this information. If it turns out that the instruction is a branch instruction then a Check fact is also asserted.

This check fact causes another rule to fire that "fixes" the branch instruction to it's original position in the basic block. This is done by making every instruction that comes before the branch a producer of that branch. This prevents the branch from being scheduled ahead of other instructions that are meant to be executed first. This is necessary because it is very rare for a branch instruction to depend upon another instruction.

Once every instruction has had it's operation data imbued into it and each branch instruction "fixed", the Runtime switches over to the Analysis phase.

4.2.2 Analysis Phase

In the Analysis phase, the Runtime is tasked with finding RAW, WAW, and WAR dependencies between instructions and then updating the dependency chains in each associated instruction to reflect this dependency. Checking for these dependencies consists of:

1. Determining if two instructions can be compared
2. Checking for the dependency

Obviously, the first condition must be met before the second condition is even tested for. Thus two instructions are compatible if:

1. The instructions are not the same
2. Neither instruction is a branch instruction
3. The first instruction selected occurs before the second instruction

If these three conditions are met then the instructions can be tested to see if there is a dependency between them. This test takes the form of checking to see if a specific set of registers from the first instruction contains any register from a specific set in the second. The three different cases are defined by the following table

Type	First Instruction	Second Instruction
WAW	Destination Registers	Destination Registers
RAW	Destination Registers	Predicate and Source Registers
WAR	Source Registers	Predicate and Destination Registers

While the Itanium 2 architecture handles WAR hazards internally it is still a good idea to check for WARs to prevent data corruption if the scheduler generates an instruction group where the instructions are not in chronological order.

It is also important to note that this test is not a simple pattern match due to the inclusion of the memory access syntax used by Itanium 2 assembler. This syntax takes the form of a register surrounded by curly braces and requires special logic in the form of a function called *contains-registerp* which tests two lists to see if any part of the second list is contained in the first. This function also performs a test to see if the memory access version of each register causes equality to occur as well. This has to happen because a specific operation may compute a memory address and then load a value from that memory address location. While it is obvious to a human that r20 and {r20} are the same register , it is not to a computer because they are different symbols. Thus it is really important that these kinds of operations remain properly ordered[5].

If the test succeeds then a Dependency fact is asserted that contains the unique id of the first instruction, the unique id of the second instruction, and the dependency type.

Each of these dependencies causes another rule to be fired that takes this Dependency information and modifies the dependency chains of the two instructions involved. The first instruction marks the second instruction as a consumer and the second instruction marks the first as a producer. Once this is done, the dependency fact is retracted.

This process of dependency fact generation and dependency chain modification continues until all dependencies have been found. Once that happens, control is then passed off to the Collect phase.

4.2.3 Collect Phase

The Collect phase is tasked with taking all of the dependency chains constructed in the Analysis phase and loading their instance names into a specific list named "Collect". Once this has occurred, control is passed off to the Schedule phase.

4.2.4 Schedule Phase

The schedule phase is where the actual schedule is constructed. The basic idea behind creating the schedule is defined by the following algorithm:

while Unscheduled Instructions Remain **do**
  Create a new instruction group G
  for each unscheduled instruction I_u **do**

[5]Especially if the register is reused multiple times.

```
      if all producers of I_u have been scheduled then
        Schedule I_u into G
      end if
    end for
    for each scheduled instruction I_s do
      for each unscheduled instruction I_u do
        if I_s ∈ Producers(I_u) then
          Remove I_s from Producers(I_u)
        end if
      end for
    end for
  end while
```

The implementation of this algorithm visits each element in the list of instructions exactly once. This is done by introducing a second list named "At" that contains the list of instructions that have already been visited. Each dependency chain in "Collect" is compared against the elements of "At" to see if the list of producers contained in the dependency chain is a subset of "At". If this is found to be true then the instruction is removed from "Collect", added to "At", and inserted into the current instruction group. This continues until there are no more elements in "Collect" and thus every single instruction has been visited. While it may seem that some instructions are visited more than once this is not the case because an instruction is considered visited only *after* it has been scheduled.

There were two major issues that had to be overcome during the implementation of this approach. The first was ensuring that dependent instructions were scheduled into the appropriate instruction group and the second was making sure that the schedule constructor did not stop after scheduling the first instruction group.

The first problem stemmed from the fact that taking a piecemeal approach to schedule construction caused "At" to be modified after each instruction was scheduled. This had the effect of causing some dependent instructions to be scheduled into the same instruction group as their producers. Solving this problem required merging the different aspects of schedule construction into a single rule. This ensures that "Collect" is not iterated over multiple times. It also ensures that the creation of an instruction group only occurs if it is needed.

The second issue, scheduling beyond the first instruction group, was solved by having a second rule that revoked and asserted the Schedule control fact if there were still instructions to be scheduled. This was necessary because the Schedule control fact was consumed after each invocation of the schedule rule and as such could not be used again. Reasserting the control fact causes the schedule operation to be fired again. This continues to occur until all instructions have been scheduled. When this happens the runtime retracts the control fact and terminates.

5 Verifying the System

Verification of a compiler optimization, such as schedule construction, is actually quite difficult to do because there is no standard mechanism to prove that the optimization is correct.

Thus two techniques have to be applied to test the System for correctness. They are:

1. Matching System output to the output generated by the expert for the same basic block.

2. Running through each rule of the Runtime to make sure that it perfectly simulates what the expert would do.

The first method requires that a large contingent of basic blocks be available for testing that are easy for the expert to schedule and verify against the output of the System. This technique has an added bonus that the differences between the System and expert output can be easily understood and used to determine the point of failure. This was the primary form of testing because it was easy to do and the amount of input was practically limitless. This is because any program compiled for the Itanium architecture can be disassembled and converted into valid basic blocks. The logic behind this approach is that if the schedule constructor works on a wide variety of input from many different sources then it must be correct. While this method is not the most formal way to ensure correctness, it is easy to do. However, when that fails it is necessary to resort to testing the System with CLIPS' watch command enabled to identify where the program is failing. This is a time consuming process that may not always yield a straightforward reason for the failure. When that happens it is necessary to verify that the failing rule is simulating the expert perfectly. If it is, then the problem stems from another source and thus the expert will be consulted until the problem can be determined. If the problem can not be found then the expert may be faulty or there may be a flaw in the production environment.

The System was verified using both of the above methods at different times. When a rule was being implemented the expert was first asked to explain how they applied the rule that needed to be implemented. Once they gave a reasonably concrete explanation, it was up to the author of this paper to implement this explanation. Once a rule was implemented it was tested using several different basic blocks where the correct result was already known. If the result of the System was the same as the correct result then the rule was denoted correct. Once the Runtime was fully implemented the same basic blocks were fed through it and the result was compared to what the expert got. If they were the same then the program was working correctly. Each basic block that was scheduled correctly served to improve the reliability of the overall System.

The System does have one weakness though. It can not safely construct a schedule beyond a basic block. While it usually works there are cases where it does not work because the intent of certain operations are not understood. More specifically, the System fails whenever the register ar.pfs is used multiple times in the same block. This register defines a stack frame and must be modified by the first and relatively last instruction within a function. This requirement was never implemented into the system because it would violate the fourth condition of the project scope. The other problem is that the rules regarding this register are somewhat strange and were not fully understood at the time of implementation. It is important to note that this is the only case where the System outright fails. Every other case has been a complete success.

6 User Interaction

The user has very little interaction with the System beyond providing it with a basic block to schedule. This is done through the use of the *block* command which initializes the register file and loads the list of instructions CLIPS. The user then has to run the *analyze* command because any rules associated with an object will automatically be fired as a result of their instantiation. The *analyze* command prevents this by asserting the Imbue control fact before invoking the *run* command. Once the System is finished constructing the schedule the result is displayed to the user in the form of instruction groups separated by explicit stops.

7 Conclusion

At the end of the day this project was a complete success. The System simulated part of an Itanium 2 processor, identified dependencies between instructions, scheduled every basic block it was given correctly, and it was constructed in such a way that it was not dependent on the specifics of the Itanium 2 microarchitecture. With the way that the schedule constructor has been written it is actually really easy to support other architectures by describing the operations and register compliment of the new architecture. The rest of the work is handled automatically by the schedule constructor. This makes the scheduler very easy to use while not limiting it's usability. This was a very eye-opening project and proved to the author that it is indeed possible to construct a parallel schedule for a basic block using an expert system.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[2] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company, 1987.

[3] J. Bharadwaj, K. Menezes, and C. McKinsey. Wavefront scheduling: path based data representation and scheduling of subgraphs. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, pages 262 –271, 1999.

[4] Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. *Embedded computing - a VLIW approach to architecture, compilers, and tools*. Morgan Kaufmann, 2005.

[5] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *Micro, IEEE*, 23(2):44 – 55, march-april 2003.

[6] S. Winkel. Optimal versus heuristic global code scheduling. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 43 –55, dec. 2007.