Critical Issues Regarding HPS, A High-Performance Microarchitecture

Yale N. Patt, Stephen W. Melvin, Wen-mei Hwu, and Michael C. Shebanow
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720

Abstract. *HPS is a new model for a high performance microarchitecture which is targeted for implementing very dissimilar ISP architectures. It derives its performance from executing the operations within a restricted window of a program out-of-order, asynchronously, and concurrently whenever possible. Before the model can be reduced to an effective working implementation of a particular target architecture, several issues need to be resolved. This paper discusses these issues, both in general and in the context of architectures with specific characteristics.*

1. Introduction

HPS is a new microarchitecture targeted for implementing high performance computing engines. Like most other proposals for high performance, we too recognize that one must exploit concurrency. Our approach for doing this is a major variation on the classical fine grain data flow theme. Our model allows a control flow architecture to be implemented by a data flow oriented microarchitecture. We call our approach "restricted data flow" because at any one time, the only data flow nodes present in our microengine are those corresponding to a restricted subset of the dynamic instruction stream. An introduction to this near model of execution and the rationale for why we believe it has great potential for high performance microengines is contained in [1].

The HPS concept is stated, intentionally, in general terms since we expect it to be a viable microarchitecture for implementing very dissimilar ISP architectures. The particular characteristics of a given target architecture will dictate many of the parameters of an HPS design. This paper explores many of the tradeoffs that are involved.

The paper is divided into six sections. Each section deals with a particular part of an HPS design. Section 2 addresses the problems inherent in decoding the instructions of the control flow ISP architecture into the data flow nodes, including the issue of whether or not a node cache is necessary. Section 3 covers the control flow issues, including the importance of branch prediction. Section 4 describes the tradeoff's involved in designing the main data path. Section 5 treats memory and I/O. Finally, Section 6 discusses two issues that are very important to a machine that implements a sequential instruction stream with a microengine that allows out of order execution: retirement and repair. We close the paper with a few concluding remarks.

2. Decoding Issues

Decoding is the process of converting the instruction stream into nodes for the HPS microengine. The decoder reads in machine language instructions of the architecture being implemented and sends data flow nodes into the main data path. It also must do some additional processing on the instruction stream that is not directly reflected in the data flow nodes in order to handle control flow changes. How much processing it needs to do depends on how control flow changes are handled, and this will be covered in the next section. In this section we will be concerned with the process of converting non-control instructions into data flow nodes, and related issues.

Originally published in *Proc. 18th Ann. Workshop Microprogramming*, 1985, pp. 109–116. Copyright © 1985 ACM. All rights reserved.

2.1. Node cache

The first question is: how much implicit sequentiality is there in the instruction stream? The more there is, the harder it will be to decode quickly. By implicit sequentiality, we mean semantics that assume sequentiality, each when it is not required by the operation being specified. One example is sequential operand evaluation (i.e. the architecture assumes operands will be evaluated in order, thus any evaluation with side effects could affect the operands that followe). Another example is a highly variable instruction format (which makes determining the location of the next instruction harder).

We must then figure out how fast the decoding process can be as a function of how much hardware we put into the decoder. This speed must be related to how fast we would like to merge the nodes into the main data path. We will probably conclude, even for instruction streams that are fairly easy to decode, that a node cache is a good idea. This is a cache that holds instructions after they have been decoded, thus relieving pressure on the decoding process itself. The size of the node cache (and therefore the hit ratio) will affect how fast the decoder has to be in order that the merging process doesn't have to wait for node generation. Thus, the cost of the decoder and the cost of the node cache must be balanced against each other.

The structure of the cache itself is also an important issue. The main problem is that a machine level instruction can potentially map into a variable number or nodes. If the architecture being implemented is fairly simple, then the variation will probably not be very great, and a simple cache scheme with a constant number of node slots per instruction could be acceptable. Even if the variation is great, this approach could still be used as long as most instructions generate a small number of nodes. Then, instructions that generate more nodes than there are slots would simply not be cachable. Of course, the time to decode a non-cachable instruction would have to be weighted against its frequency of occurrence.

If the variation in nodes per instruction is great, and a significant portion of instructions generate many nodes, then a two table approach might be more practical. One table would contain instruction entries, each of which would contain the instruction address and a pointer into another table. This second table would contain the actual nodes. The first table would have the associative search mechanism, while the second table would be where the actual node would be stored. This scheme has the disadvantage that replacement is more difficult. The second table could become fragmented, causing space to be wasted unless some kind of compaction mechanism is used. Also, the two table approach incurs an extra level of indirection when nodes are retrieved. The replacement algorithm is also something to be considered. LRU would seem to be the most favorable, but in the case of the two table scheme, a variation that reduces fragmentation might be better. Also, a cheaper to implement algorithm might be almost as effective.

Another important issue is what to cache. This has to do with on what the nodes for an instruction depend. If the nodes depend only on the instruction itself, then everything is cachable and the nodes can be generated easily. If the nodes depend on the instruction plus other constants in the instruction stream, then everything is still cachable, but the node generation process is a little more difficult. This situation arises for example in relative addressing modes where the current PC is added to an offset, or for instructions that have literals in the instruction stream. A third situation arises when the nodes for an instruction depend on the instruction stream as well as variables not present in the instruction stream. In this case, some of the nodes may not be cachable, but it's not clear that these nodes are definitely not cachable. For example, consider a CALL instruction that gets a register save mask from the subprogram being exiled. If the destination is static, and if the code is not self-modifying, then the same CALL will always save the same registers, so it would probably be acceptable to cache the nodes that save the registers. There may be cases where the nodes can definitely not be cached, such as a CALL instruction where the destination is provided through a register, but the cache still might be able to provide useful information.

Finally, consider the question of caching a process identifier along with the nodes. On a multiprogrammed uniprocessor, it may be desirable to allow the node cache to retain its contents between process switches, so each instruction would have to be associated with a process. There could also be a reserved section for system notes as opposed to user notes. (Note that if HPS were implemented to allow simultaneous execution of multiple instruction streams, then process identifiers would not only have to be in the note cache, but attached to each node in the machine.)

2.2. Predictive decoding

Another decoding issue has to do with how we determine what to fetch from the instruction stream and decode. If the decoding process is very fast and no node cache exists, then the decoder will closely follow the merger. There would not be much of a question of what to decode next. However, if decoding is slow and the node cache is large, determining how to predecode intelligently isn't as simple. One idea would be to decode the first few instructions of the side of a branch that is not taken. This would be especially helpful if the branch predictor could provide a confidence level, indicating the expected probability of a wrong guess. Then, the decoder could decide how much to decode down each side of a branch based on this confidence level. If the frequency of control nodes in the program being run is low, then predictive decoding would probably not make much difference, but if the frequency is high, it could be quite significant.

3.3. Node generation optimality

The next decoding issue we will discuss is the optimality of the node generation process. The more optimally we generate nodes, the faster the machine will execute an instruction, but the slower and more expensive the decoder will be. It may be counter-productive to handle a special case that reduces the number of nodes by one in every million. It might seem that determining the best way to generate nodes for a particular instruction is simple, but consider instructions with complicated operand evaluation. If the operands within an instruction are to be interpreted sequentially and they successively modify the same register (such as for auto-increment), then more than the optimal number of nodes could be generated if the operands are decoded independently. The mechanism for coordinating them could be complicated, however. Another example of sub-optimality occurs when the value of an operand affects the nodes generated (e.g. when a literal operand would simplify the nodes generated). This could be important for instructions that are highly general, but for which most instances of them don't use much of the generality (e.g. if there are some operands that are literal zeros in the instruction stream that significantly simplify the instruction).

2.4 Special cases of instructions

Another issue is what to do for certain special eases of instructions. First we will consider the case of instructions that stall. A stall is defined as that time when the decoder must wait for the main data path or the memory and I/O unit until it can proceed. There are two basic types of stalls: stalls for the next instruction address, and stalls for node generation. Stalls for the next instruction address are cases where the point of next execution cannot be determined (or even predicted) until a node is distributed. These types of stalls will be discussed more thoroughly in the next section. Stalls for node generation are cases where the nodes to complete an instruction cannot be generated until a node is distributed (an example is a CALL instruction that needs to wait for a register save mask before it can generate the nodes that save the registers). Most architectures have at least some instructions that cause the first type of stall, but the number of instructions that cause the second type of stall can vary greatly. More complex architectures, that are characterized by control points inside an instruction, are more likely than simple architectures to have instructions that cause node generation stalls.

One issue that comes up for handling instructions that stall is how to handle the cache. If it has been determined that the instruction should cache all of its nodes, then the problem is only one of putting the two groups of nodes (i.e. the nodes generated before the stall and the nodes generated after the stall) under the same instruction entry. If the instruction cannot cache its post-stall nodes, then it could still cache the pre-stall nodes, but it might be simpler to not cache the instruction at all. Another issue is the interaction between the decoder and the rest of the machine. As far as the decoder is concerned, it must generate a node and wait until the result gets distributed before it can continue decoding. This probably means that the decoder needs to watch the distribution bus. But suppose that an instruction depends on a register and that register is ready. Then, the decoder would generate a redundant move node and wait until it is distributed unless is has access to the registers directly. The first option generates spurious nodes and the second option makes the mechanism more complicated. Another question is should the nodes that the decoder is waiting on be given extra priority, and if so, how does this mechanism work (we will discuss this further in the main data path section).

Another special case that some architectures may have is instructions that operate on large blocks of memory (for example block move, pattern search, etc.). How should the nodes be generated for instructions such as these? The decoder could simply generate all of the nodes, but in the case of a pattern search, it would have to stall between each pair of memory nodes. Also, the cache could be wiped out, so it would be more desirable to not cache these instructions. A better solution might be to implement the move as a loop, visible at the machine instruction level, so the branch prediction mechanism could help out on each iteration. If this were done, then caching the instruction would be a lot simpler. Alternative ways of handing these instructions will be covered in the memory and I/O section of the paper.

3. Control Flow Issues

In this section we will discuss the issues related to handling the flow of control in the machine. Recall that there are no control nodes. The only nodes that are merged into the data path are data flow nodes. This means that the decoder must handle the changes in flow of control that are specified by the machine language. In this section, we will use the term “branch” to mean any change in the flow of control of the machine. There are two basic issues here: 1. what are the conditions of the branch? and 2. can the destination(s) be determined statically? If the destination address or addresses are known to the decoder, then it can make a prediction on the outcome and proceed to decode more nodes. (Unless we have a many-way branch, for which the outcomes are uniformly distributed, making the payoff for predicting overweighed by the cost of bracing up). If, however, the destination address is unknown, we can only stall and wait until it becomes known, regardless of the conditions of the branch. (We could, of course, predict a destination address, and with a history mechanism of some sort this might be worthwhile, but for many cases, any prediction at all would just be a stab in the dark.)

Therefore, we can group branches into two categories: those which we can effectively predict, and those which we cannot. Exactly what it means to “effectively” predict a branch is, of course, somewhat nebulous but it would depend on the quality of the branch prediction algorithm, the cost of the repair mechanism, as well as the behavior of the program being run. Examples of instructions that will generally be predictable are simple two-way branches where the destination for both sides of the branch are static. Examples of instructions generally unpredictable are return instructions which have to pop the top of the stack to determine the next address, and jump indirect instructions that have to read the contents of a register or a memory location to determine the next address. The decision of what to predict and what to let stall, as well as the characteristics of the branch prediction algorithm, are fairly critical issues. The objectives are to predict as correctly as possible to minimize repair and to cover as many cases as possible to minimize stalls. A version of HPS has been simulated using an auto-correlation branch predictor that detects pattern in the behavior of branches, and has been shown to be quite effective.

Given that a branch has been predicted and the decoder has gone on to decode more instructions, how does the confirmation mechanism work? One alternative is to put the mechanism in the decoder. Then, the decoder would associate a particular node with the branch condition so that when that node gets distributed, the branch can be verified. In the case of branches on condition codes, the node is simply the last node which generated the particular condition code. This means that the decoder probably needs access to the condition code alias table. For more complicated branches, where the condition must be computed, the node that does the computation will be the one that the branch predictor watches for.

Another scheme would be to have branch nodes that are merged into the main data path just as any other node, and to have a branch function unit that executes the branch nodes. The branch nodes would take their inputs from the nodes that the decoder was waiting for in the previous scheme. With either method, when a branch prediction error is detected, the decoder, the predictor, and the main data path, must be notified. Some sort of mechanism is necessary to associate instances of branches to node indexes used by the main data path so that the correct repair point is referred to. The main data path must back up the node table and the decoder must redirect its decoding (the repair mechanism will be discussed more fully in section 6).

4. Main Data Path Issues

The main data path is that portion of the machine that contains the active nodes and the fraction units that are executing them (except for memory and I/O nodes). The node tables hold the active nodes, and the three processes associated with the main data path are merging, scheduling, and distribution. First we will talk about the structure of the node tables and then we will talk about each of the above processes.

4.1. Node table structure

One of the main ideas of HPS is that more than one node can become active on each cycle, and more than one node can be executed on each cycle. A logical way to achieve this throughput is to partition the node tables. If there is only one node table, then the bandwidth required would be enormous. If there are several node tables, then a single node can be merged into each node table on each cycle, and we will still have several nodes merged per cycle. The main question that arises is how to divide up the node tables with regard to the function units. A straightforward way is to divide the node tables by the function being performed. So, for example, there would be a fixed point node table, a coating point node table, a memory node table, etc. Then, a particular node could only be merged into one particular node table. There are also other possible ways to partition the node table. There could be several node tables that serve the same type of function unit, or the node tables could overlap only partially in functionality. This would mean that a particular node could be merged into more than one node table, making the merging process more complicated. This partitioning of the node tables gives rise to what we call the “multinodeword.” A multinodeword is a group of nodes that are all merged together, each part of it going to a different node table.

Another issue relating to the node tables is whether or not the node values are stored in the node table, or in a separate structure. Each node in the node table must by necessity hold the tags and the ready bits for the nodes that it depends on, but it doesn't have to contain the actual values. A possible organization would be to separate the values from the node table in a "value buffer." Then, nodes would get their operands from the value buffer and their results would go back into the value buffer. The advantage of this scheme is that it makes the node tables smaller, and since they must have associative memories, this could be important. The main disadvantage is that when a node is determined to be ready and it is read from the node table, the tags must then index into the value buffer to get the values. This gives an extra level of indirection, which could slow the machine down.

4.2. Merging

Merging is the process of taking nodes from the decoder and installing them into the node tables. The amount of processing that the merger must do on the nodes depends on several things. One is the question of how the nodes are grouped into multinodewords. If the cache stores nodes in a free format, then the merger must group them into multinodewords before they go into the node tables. This has the disadvantage of putting a lot of pressure on the merger. Alternatively, the nodes could be grouped together between the decoder and the cache, or within the decoder itself. Having the decoder generate the nodes already in multinodeword format might not be practical for complicated architectures. It might be better to have the decoder generate the nodes and then have a post-processor put them into multinodewords. Another option, if the architecture being implemented involves direct execution of nodes, is to have the nodes grouped together by the complier before they ever reach the decoder. This would relieve the most pressure from the hardware.

Another merging issue is the handling of the register alias table (or RAT). Recall that the RAT is the table that stores the node tags that are forwarded in order to remove false dependencies as in the Tomasulo algorithm [2]. Since there are potentially several read/write accesses per multinodeword (each node could read from two entries and cause a write to one), the RAT most be a multiported memory. The complication could be reduced if instructions were placed on how register reads and writes could be used in multinodewords, but this could slow down the machine. This brings up the store important issue of the interpretation of two nodes within the same multinodeword, and two nodes within the same instruction that have register conflicts. If the RAT is updated for register writes between each multinodeword, then a node that writes into a register and a node that reads from the same register could be placed in any order within a multinodeword, but not within an instruction. If, alternatively, the RAT is only updated for register writes in between each instruction, then these two nodes could be placed in any order within the instruction. This would allow more complicated instructions to be implemented simpler (e.g. swap could be done with two nodes instead of three). The disadvantage of this scheme is that it makes the RAT more complicated.

Another issue is the mechanism for sending literals into the data path. A literal present in the instruction steam that is needed for computation node must somehow be entered into the data path. If the operand values are stored with the nodes in the node table, then the merger could simply insert the literals in the operand fields of the nodes before they are merged. This requires that the literals be known before the nodes that use them are merged. Also, a literal wouldn't need to have a tag associated with it, since it is never distributed. Another alternative would be to associate tags with literals and then distribute them normaly and let every node that needs them grab them. Note that the literal could then be distributed at any point in the instruction and the scratch pad alias table would handle things. This scheme requires the merger to have source access to the distribution bus, which might complicate things significantly.

4.3. Scheduling

Abstractly, scheduling is the process of determining which nodes are ready, selecting a subset of them to be executed, and sending them to the function units. The simplest and easiest to understand algorithm is oldest node first. Since the nodes must by necessity be merged into the node table in the order that they were decoded, it should be fairly simple to select the oldest node and send it to the function unit to be executed. This is not necessarily the optimal algorithm, of course, but it probably performs close enough, and at a cost far below a more complicated algorithm, to be a win. Also, there are other factors besides optimality that must be considered. For one thing, we don't want old nodes hanging around in the machine too long because then we can't retire instructions soon enough. Another thing is that the oldest nodes have the lowest probability of having to he undone because of a branch prediction miss or an exception. Actual simulations will probably be the only way to really settle the scheduling algorithm question, and of course it could vary significantly from one architecture to another.

An issue that was raised in the decoding section of the paper was the question of giving certain nodes high priority. It would seem that if the decoding process is waiting for a node before it can continue, that node should be given higher priority. But should it be given the highest priority? And what about nodes that confirm branch predictions? Should they be given high priority also? It would seem that since confirming branchs and minimizing stalls are fairly important to the overall speed of the machine, some sort of priority scheme should be devised, but it is not clear how it should work.

4.4. Distribution

Distribution is the process of sending the computed results back to the node tables as tag-value pairs so that the ready bits can be updated and the values can be stored for subsequent use. There are many function units and each function unit could in general be sending its result to any node table. So, the most general distribution scheme is to have one bus for each function unit and to have each bus go to every node table. The mechanism could get expensive since we need to handle the case where every fraction unit wants to update every node table. If there is a value buffer, then the problem gets a little bit better because then only the tags need to go to every node table. The values would go to the value buffer, which can be partitioned to allow multiple writes. It some restrictions were placed on the nodes, such as where they could take their inputs from, then perhaps the distribution could be made easier, but this is at the cost of requiring more nodes in some cases. We really haven't developed an effective alternative mechanism to total distribution, but it is clear that this may be one of the major bottlenecks of the machine.

5. Memory and I/O Issues

The memory and I/O unit is that portion of the machine that holds all the nodes that access memory and I/O devices, and interacts with the memory and I/O systems of the machine. First we will discuss a couple of fundamental problems that this unit must cope with and then some overall issues. In this section, we will use the terms "memory system" and "I/O system" to refer to those parts of the machine concerning memory and I/O operations respectively that are not part of the memory and I/O unit (e.g. the data cache, the translation mechanist, main memory, etc., and the I/O devices, I/O control register, etc. respectively).

5.1. The unknown address problem

One of the main ideas in HPS is to allow out of order execution of nodes but to preserve the sequential semantics of the instruction stream. This means that the result of executing memory reads and writes must be the same as if all nodes were executed in the order that they were merged. If the addresses that the memory operations use are static (i.e. they are all known to the decoder) the problem of resolving dependencies is quite simple. It would just involve comparing the addresses and determining which memory operations conflict. Memory operations to the same address could be resolved in the same way that arithmetic operations to the same register are. Furthermore, this would all be known when the nodes were merged. However, if the addresses of the memory operations are determined dynamically (i.e. they cannot be determined by the decoder) then the problem becomes more complicated. This would be the case for many memory nodes, for example indirect and indexed accesses. The dependencies between the memory operations would have to be determined as the information becomes available.

Consider only simple memory read and memory write nodes with the following definitions: each read node has one input (the address) and one output (the data) while each write node has two inputs (the address and the data) and no outputs. The inputs to each node may be known or unknown and an unknown input may become known at any time. Each read node can therefore be in one of two states: address known (AK) and address unknown (AU). Similarly, each write node can be in any one of four states: address known - data known (AKDK), address known -data unknown (AKDU), address unknown - data known (AUDK) and address unknown - data unknown (AUDU). We will call nodes in the AK and AKDK states "ready," meaning that they are ready to be executed baring dependencies on other nodes. So, the question becomes: under what circumstances can a ready node become executable?

An executable node is one which can be "unexecuted," or sent to the memory system to be performed. In the case of reads, this simply means that a request is sent to the memory system. In the case of writes, execution means that the write buffer is added, but the operation is not actually sent to the memory system until the instruction retires. The simplest and lowest performing approach would be to execute the nodes in the order that they were merged. On the other extreme would be a method which detects every dependency in every possible circumstance and executes nodes as soon as possible. We will now present the optimal conditions under which a node can be executed (i.e. the least constraining dependency resolution possible). As we will see later, however, it may be desirable to make more restrictive conditions in order to make the implementation more practical.

It only makes sense to discuss the condition for executing ready nodes since other nodes couldn't possibly be executed and it is also the case that only the read nodes need to be considered, since a write node can be "executed" as soon as it becomes ready. So, we are left with the question of when a ready read is executable and the answer can be expressed algorithmically as follows:

```
Find the youngest active write node that is older than the read nodes:
    If it's an AKDU or AKDK write with a different
        address, find the next oldest write node and
        repeat this step
    If it's an AKDK with the same address, stop—the
        node IS executable and this AKDK has the
        data
    If it's an AKDU with the same address, stop—the
        node IS NOT executable, but it is only waiting
        for the AKDU to become an AKDK
    If it's an AUDK or AUDU, stop—the node IS
        NOT executable
If there are no (more) active writes, the node IS
executable.
```

Now we will discuss two implementations of the memory data path. The first one is a fairly straightforward approach that is easy to implement but isn't fully optimal (i.e. it may hold back a node in certain circumstances that could actually be executed). The second method achieves full optimality but is more expensive to implement. These are only two initial approaches to the memory unit implementation, and should serve to illustrate some of the issues.

The first scheme is a basic extension of the register dependency resolution algorithm used in the merger. If the memory is considered to be a very large register set, the extension is clear. The memory alias table (or MAT) is analogous to the register alias table. It must be associative on the memory address, since there can't be an entry for every memory location, but its function is the same. The only complication is that the MAT will only contain the proper information for a particular read node if there are no AUDK's or AUDU's that are older. This is handled by the memory node table. It holds nodes until they can be released to update the MAT in the ease of writes, or access the MAT in the case of reads.

The second scheme is a more complicated one but achieves fully optimal dependency resolution. It is too complicated to go into in full detail here, but the basic idea is to incorporate both the MAT and the memory write buffer into a single structure. A dependency matrix is created that relates each operation to every other operation. Each read node starts out with a mask register, representing a row in the dependency matrix. As addresses become known, bits are cleared in the mask until either all bits have been cleared or a single one remains and that one corresponds to a known address and a known data value. At this point the read can be executed. There would be a high level of intercommunication necessary between different parts of the memory unit with such a scheme, but this may be practical to implement, depending on the size of the node table, etc.

5.2. The partial dependency problem

The above discussion about resolving the dependencies between memory node has assumed that there are no partial dependencies, but in most architectures, they can occur. A partial dependency is defined as a dependency between two nodes when there is not a total overlap between the pieces of memory that each node refers to. This can occur for two reasons: memory nodes are allowed to refer to different sized pieces of memory, and memory accesses are not aligned to the size of the memory piece. In many architectures, both of these are allowed to happen, and most architecture allow at least one, so this is a problem that is likely to be important. There are two aspects to this problem: how to resolve the dependencies and determine which nodes are executable, and how to actually execute the notes (since a node could potentially need to get its value from more than one place). These two aspects are interrelated, and in general the solution to how the nodes are executed will have a large influence on how the dependencies are resolved.

One approach might be to have the decoder remove all partial dependencies by aligning all memory operations and forcing them to all be the same size. This has the disadvantage that the decoder must know what the addresses of every memory operation are before the nodes are created. Although this would make the memory unit a lot less complicated, it would probably slow down the machine significantly.

Given that the problem is not solved by the decoder, one issue is where in the machine partial dependencies are pieced together. If the memory unit is sufficiently complicated, it could forward data on a piecewise basis. Note that a node could potentially need parts of its value from several other nodes and from the memory system. This means that the memory unit would have to issue the memory operations to retrieve the needed parts and when they arrive it would have to assemble the data and distribute it. This type of scheme would necessitate a very complicated dependency resolution mechanism.

Another possibility would be to have the memory system deal with the assembly of these nodes. This would only work if all the nodes that another node is partially dependent on have been issued to the memory system (i.e. the associated instruction has been retired).

The scheme would allow the dependency resolution mechanism to be a little bit simpler, since it would only be necessary to detect a conflict, not to determine the extent to which it overlaps. A disadvantage of this scheme is that it doesn't result in optimal note execution. A node could be held up even if its data is ready because a node that it depends on hasn't been retired. The penalty for such sub-optimality must be weighed against the frequency of occurrence and the cost of implementing a more optimal scheme. And, these factors are very dependent on the architecture being implemented and even the application of the architecture. In general, the partial dependency problem is quite complicated and its solution involves many tradeoffs. These have only been touched on here, but it should be clear that there are many possibilities that have yet to be fully explored.

5.3. I/O issues

The handling of I/O operations bring up some interesting issues. We have been assuming above that there are no problems with executing memory reads out of order. We have a model of what memory reads and writes do, and we have used this model to improve the speed of the machine. In the case of I/O operations, this model needs to be re-examined. It is probably the case that I/O reads (as well as writes) must be executed in the same order that they would be if the instruction stream were executed sequentially. This means that the same methods that are applied to memory nodes cannot be applied to I/O nodes.

If I/O operations have their own opcodes, then the problem can be handled relatively easily. The decoder would simply generate I/O nodes that would be recognized by the memory and I/O unit as such. These nodes would be held until the instruction retires in the same way that memory write nodes have to be held. A slight problem arises if an instruction reads from I/O and sends the result to memory. Deadlock could result if the I/O read is waiting for the memory write to execute. The solution is to execute I/O nodes under the condition that all main data path nodes have executed and to execute memory writes under the condition that all other nodes have executed (this will be covered in more detail in section 6).

The problem becomes much harder if the I/O is memory mapped. If I/O addresses are easily recognizable, then the memory and I/O unit could determine the type of node when the address is determined and treat each type separately. However, if the addresses are not easily distinguishable, this won't work. This case arises in virtual memory machines even if the physical addresses can be easily recognized. However, at some point an access must be resolved into memory or I/O, so we could have the system that first determines this report back to the memory and I/O unit if the access is to an I/O device. The memory and I/O unit could then handle it as appropriate. This requires the memory and I/O systems to be more complicated, but in the absence of an architecturally constrained definition of an I/O operation, it is about the only solution (other than executing all memory and I/O nodes sequentially).

5.4. Other memory and I/O issues

Another issue for the Memory and I/O unit that was alluded to earlier is the question of how to handle instructions that operate on large blocks of memory. Some architectures don't have these instructions, but they are worth investigating for those that do. The simplest example of such an instruction is one that moves a block of memory from one location to another. At some level, an instruction like this must be broken down into a series of reads and writes. This could occur at the decoding level (the decoder generates the sequence of nodes), at the merging level (the decoder generates a loop, which is turned into a sequence of nodes before merging), at the memory unit level (the memory unit gets a single node which is turned into a sequence of reads and/or writes to the memory system) or at the memory system level (the memory unit issues a single command to the memory system which then performs the function on its own.) There are tradeoffs for placing it at each level, but one thing to consider is that if the instruction involves more than just a block move (for example a pattern search) then placing it closer to the memory system gets more complicated.

6. General Issues

6.1 Repair

Repair is the backing up of the machine to a previous point and it is done when nodes have been merged that shouldn't be executed. This previous point can be before any instruction that has not been retired. The amount of hardware that is devoted to repair (and thus the speed that it can take place) will be dictated by the frequency that it is necessary, and this is very dependent on the architecture. Repair is necessary for three reasons: a branch prediction miss, an exception, and an interrupt. It will probably be the case that branch prediction misses are much more common than either of the other two causes, so we may be able to take advantage of this. Branches don't occur every instruction, so while we must be able to back up between any two instructions in the active window, it only may need to be fast to back up to the instruction before a branch.

Consider what has to be done to complete a repair operation. First of all, the node tables have to be backed up. This involves invalidating all nodes merged after the violation point. Since the nodes are stored in the order that they were merged, this can be accomplished fairly simply. However, it does mean that we need to keep node table indexes for each instruction. The register write buffer and memory write buffer also need to be backed up. This could be handled analogous to the note tables since they are ordered by instruction.

The RAT and MAT must also be restored, and this is a bit more complicated. In order to restore the alias tables quickly at any instruction boundary in the instruction window, a lot of back up copies may be required. An alternative would be to only store alias table copies for the pending branch predictions. Then, we could just let the node tables train for exceptions and interrupts and wait for the alias tables to clear. A disadvantage of this scheme is that it may limit us as to how many branch predictions may be pending at a time.

6.2. Retirement

Another general issue is that of how retirement is handled. The first question is how you know that an instruction is done and can be retired. Since nodes are stored in merge order, we really just need to know the oldest unexecuted node for each node table. This information, along with information about each instruction, will tell us if the oldest instruction in the window is ready for retirement. It is not clear how this information should be collected and sent to the appropriate place, however. There probably needs to be a retirement control unit that coordinates all the information.

Several things must get done when an instruction retires, and exactly how to handle these is also an important question. The register write buffer and the memory write buffer hold information that must be processed when an instruction retires. The processing of the memory write buffer consists of sensing the writes to the memory system. An interesting question is how to handle the case that these writes are not completable. If a write causes some sort of exception, then we can't have destroyed anything that we need to repair. It even becomes more involved if we include I/O operations. As we discussed above, it may be necessary to execute I/O reads before memory writes, but after every other note has executed. Therefore, an I/O operation causing an exception also must be considered. It would seem that the machine needs a state after all CPU nodes have executed, but before I/O operations have executed and another state after I/O operations have executed but before memory writes have executed. Only after memory writes have executed is the instruction really "retired." (This last state might be able to be eliminated if we can guarantee that writes can be completed before they are issued.)

7. Conclusion

We have investigated some of the major issues that arise in designing an HPS implementation for a particular architecture. There are still many open questions about how to handle certain problems, which should not be unexpected since HPS is still in the early stages. But this paper should give the reader an indication of where the major questions lie and how they could be approached.

8. Acknowledgment

The authors wish to acknowledge first the Digital Equipment Corporation for their very generous support of our research in a number of ways: Bill Kania, formerly with Digital's Laboratory Data Products Group, for providing major capital equipment grants that have greatly supported our ability to do research; Digital's External Research Grants Program, also for providing major capital equipment to enhance our ability to do research; and Fernando Colon Osorio, head of Advanced Development with Digital's High Performance Systems/Clusters Group, for providing funding of part of this work and first-rate technical interaction with his group on the tough problems. Part of this work was sponsored by Defense Advanced Research Projects Agency (DOD), Arpa Order No. 4871, and monitored by Naval Electronic System Command under Contract No. N00039-84-C-0089. We also acknowledge the other members of the HPS group, Chien Chen and Jia-juin Wei, for their contributions to the HPS model as well as to this paper. Finally, we wish to acknowledge our colleagues in the Aquarius Research Group at Berkeley, Al Despain, presiding, for the stimulating interaction which characterizes our daily activity at Berkeley.

References

[1] Patt, Yale N., Hwu, Wcn-mei, and Shebanow, Michael C., "HPS, A New Microarchitecture: Rationale and Introduction," Proc. 18th Int'l Microprogramming Workshop, 1985.

[2] Tomasulo, R.M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J. Research and Development*, Vol. 11, 1967, pp. 25-33.