On Tuning the Microarchitecture of an HPS Implementation of the VAX

James E. Wilson, Steve Melvin, Michael Shebanow, Wen-mei Hwu, and Yale N. Patt

Computer Science Division
University of California, Berkeley
Berkeley, CA 94720

Abstract

The HPS Microarchitecture has been developed as an execution model for implementing various architectures at very high performance. A considerable amount of effort has gone into the use of HPS as a microarchitecture for the VAX. In this paper, we describe our first full simulation of the microVAX subset, and report the results of varying (i.e. tuning) certain important parameters.

1 Introduction

HPS (High Performance Substrate) is a microarchitecture for implementing high performance computing engines. It exploits concurrency by using a restriction of classical fine granularity data flow [6]. This is a part of the Aquarius project, whose goal is to obtain enormous improvements in computer performance, in part by exploiting concurrency at all levels of transformation. HPS represents our approach to dealing with concurrency at the microarchitecture level.

This paper builds upon the work of two previous papers. The first paper [4] covered the initial design for an HPS implementation of the VAX. It described the functional units necessary to implement the VAX, and provided some encouraging preliminary results. The second paper [7] explained details of how to generate data flow nodes from a sequential VAX instruction stream. This paper covers the implementation of a simulator for the microVAX subset. Results obtained from running a set of benchmarks on the simulator are presented.

Section 2 of this paper describes the HPS VAX specification, including some discussion of the parameters to be tuned. Section 3 describes the simulator model. Section 4 reports simulation results obtained by tuning certain parameters, and provides an analysis of these results. Finally, section 5 offers some concluding remarks.

2 HPS/VAX Implementation

2.1 HPS Execution Model

The HPS execution model is a restriction on classical fine granularity data flow, hence we are calling it "restricted data flow". However, there are substantial differences between our model and the classical fine granularity data flow engines of Dennis[3] and Arvind[2]. The most important difference is that our data flow oriented microarchitecture is implementing a conventional control flow oriented ISP architecture. We define the "active window" as the set of ISP instructions whose corresponding data flow nodes are currently part of the data flow graph which is resident in the microengine. Consequently, in our model, only a small subset of the entire program is present in our microengine at any instant of time. As the active window moves through the dynamic instruction stream, HPS executes the entire program.

An abstract view of HPS is shown in Figure 1. The static instruction stream is fetched, with branch prediction, to create the dynamic instruction stream. Branch prediction is necessary to prevent stalls due to branches. Incorrect predictions are handled by the checkpoint/repair mechanism.

The decoder takes instructions from the dynamic instruction stream and generates data flow nodes for the instructions. Only instructions within the active window will have nodes active in the machine, unlike classical data flow in which the entire program must be in the machine. This is why we refer to HPS as restricted data flow. Parallelism that exists within the active window will be fully exploited by the data flow microengine.

The merger takes nodes generated by the decoder and merges them into the entire data flow graph for the active window. A generalized version of the Tomasulo algorithm[8] is used to resolve all data dependencies existing between the new nodes and the nodes in the machine. The merger uses information stored in

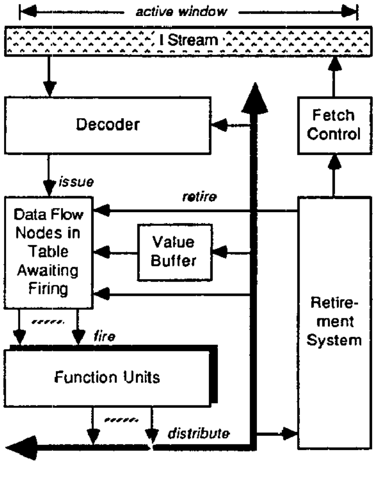


Figure 1. HPS Overview

the Register Alias Table (RAT) and the Scratch Pad Alias Table (SPAT) to help resolve these dependencies. The merged nodes are then stored into the node tables.

The scheduler searches the node tables for nodes that are ready to fire. A node is ready to fire when all of its operands are ready. The oldest node in each table with all of its operands ready is sent to the function units for execution. A typical implementation could have five function units: 2 fixed point, 1 memory, 1 branch, and 1 floating point. This scheme transparently handles multiple function units and out of order execution.

After execution, the results of each node must be distributed. To do this, each function unit gates its result to its own distribution bus, which is connected to all the node tables and alias tables. A value is put on the bus along with a tag, any node or alias table entry that has a matching tag will load the value and set its ready bit.

Finally, all instructions go through the retirement stage. Instructions are sequentially retired, after all of their nodes have been executed. This enforces the sequential semantics of the target architecture, allowing the HPS implementation to handle exceptions and interrupts identically to traditional implementations (i.e., precise interrupts[1]). The retirement stage interacts with the checkpoint/repair mechanism to achieve this.

2.2 Functional Elements

This section covers the functional elements of the HPS VAX implementation.

2.2.1 Decoder

The decoder takes an instruction and generates the data flow nodes necessary to execute that instruction. This mainly involves inserting operand values into known templates of nodes. The operand specifiers are decoded first, the nodes generated depend only on the addressing mode. Next, the nodes for the opcode are generated. For most instructions, this just involves taking the known template of nodes, and inserting operand values into the appropriate locations.

For some instructions however, the data flow nodes needed depend on the values of the operands. For example, the PUSHR instruction pushes registers onto the stack as indicated by a register mask. Since a node must be generated only for each register to be pushed, the nodes cannot be generated until this value is known. If this mask is not an instruction stream literal, then the decoder will stall until this value becomes known. This is referred to as a node generation stall.

Another example that causes stalls are CALLS and RET instructions. Since the next instruction address will come from the stack, the decoder may have to stall if this value is not known by the time it has finished decoding this instruction. This is referred to as a branch destination stall.

2.2.2 Node Cache

The performance required from the decoder can be reduced by storing previously decoded instructions in a node cache. A hit in the node cache means an effective decode time of 1 cycle. Unfortunately, this does not solve all of the problems, because not all instructions can be cached. In general, an instruction that causes a stall can not be cached. For example, the PUSHR instruction can not be cached if the register mask is not an instruction stream literal. In this case, the nodes needed for two different executions of this instruction will be different if the mask changes. We have measured the frequency of occurrence of these stall conditions[7]. Fortunately, in the case of the PUSHR instruction, in all benchmarks measured, the register mask was always an instruction stream literal.

2.2.3 Value Buffer

Each node in the machine has a unique tag associated with it. These tags are used to uniquely identify every value in the machine, and can be thought of as identifying the arcs in a data flow graph. These tags allow a node to use the value of a previous node by referring to the tag of the previous node. All values in the machine are stored in the value buffer (VB). The tag is an index into the VB. Note, the tags wrap around at the end of the VB. The size of the VB is a function of the window size. For a window size of 8 instructions, an eight bit tag is sufficient.

A special area (the low 32 locations) of the VB is reserved for the permanent values of the dynamic registers. Since the VB is a circular buffer, all values are eventually overwritten. To prevent the loss of register contents, they are stored in this area. When an instruction retires, registers modified by the instruction must have their values copied to this area.

2.2.4 Register Alias Table

The register alias table (RAT) manages all data dependencies involving the registers. For each register, the RAT contains a ready bit and a tag. If its ready bit is set, then the register's value is in the value buffer. If its ready bit is clear, then its value is not yet known.

The machine registers are divided into two categories, static registers and dynamic registers. Static registers are those which change infrequently or must be known to the decoder, while dynamic registers are those which are changed frequently and don't have to be known by the decoder. For example, most of the Program Status Longword (PSL) must be known by the decoder at all times, and hence is implemented as a static register, whereas the general purpose registers are implemented as dynamic registers.

Dynamic registers are represented by entries in the RAT. The RAT aliases a tag to each dynamic register, so that these registers can be referenced even if their value is not currently known. Static registers, on the other hand, force the decoder to stall when they are written to. The trade off is that much more hardware is needed for dynamic registers than static registers.

The RAT has entries for the 16 general purpose registers, the four condition codes, and the five architecturally defined stack pointers. Even though the condition codes are part of the PSL, they need to be implemented as dynamic registers as most instructions modify them. Making the stack pointers dynamic registers prevents the Change Mode (system call) instructions from generating stalls.

2.2.5 Scratch Pad Alias Table

The Scratch Pad Alias Table (SPAT) is needed by the merger to correctly handle instructions that take multiple cycles to merge. For every node generated by the decoder for the current instruction, there is a slot in the SPAT that corresponds to it. Each entry in the SPAT is a single bit that indicates whether or not the value of the corresponding node is valid. Note that this means the SPAT must also be loaded from the various distribution buses. When merging a node that depends on a previous node (within the same instruction), the merger checks the SPAT to see whether or not the operand ready bit should be set.

Consider the case of an instruction which takes two cycles to merge. On the second merge cycle, there will be nodes that refer to nodes merged during the first merge cycle. Usually, these nodes will not have been executed, but this is not always true. There could have been a stall due to a full active instruction window, for instance. During this stall, nodes from the first merge cycle could have executed before the second merge cycle occurred. If the rest of the nodes were subsequently merged with their operand ready bits clear, they would never execute as they would be waiting for values which had already been distributed. The SPAT solves this problem by always indicating which nodes of the current instruction have been executed.

2.2.6 Merger

The merger takes nodes from the decoder and resolves all dependencies between the new nodes and the data flow graph for the active window. This involves assigning unique tags to each node, and making sure that each internodal reference uses these tags. Each register read is replaced by the tag and ready bit for that register from the RAT. Each register write results in the RAT tag for the appropriate register being set to the tag of the node modifying it. Also, the ready bit for the register is cleared. This operation is done at the end of the cycle, so that all nodes merged during the same cycle will see the same tag for each register. The condition codes are handled in the same way, each node that modifies a condition code updates the RAT entry for that condition code.

2.2.7 Node Tables

After the merge process is done, the nodes are in the node tables. The node tables hold nodes while they await execution. The node table can be represented as a circular buffer, with new nodes being inserted at the tail. Each node table entry has a valid bit to indicate whether or not it contains a node. The nodes themselves have two ready bits, one for each operand. Every cycle, the node closest to the head that has all operands ready will be sent to a function unit for execution. This entry can be determined by a simple priority circuit.

There is a node table for each function unit. This simplifies the scheduling of nodes, as at most one node per cycle needs to be read from each node table. This, however, does create the problem of how to store nodes in the node tables for which identical function units exist.

2.2.8 Function Units

There are four types of function units in the simulator: Fixed Point, Floating Point, Memory and Branch. (Floating Point functions have not been implemented.) The Fixed Point function units (identical if more than one) have 24 operations defined. These include the expected arithmetic operations, special operations for doing double word (sic) multiplies and divides, and some special operations for manipulating the condition code fields of values. (It should be pointed out that each "value" in the machine is actually a 32 bit value plus a four bit condition code.) The memory function units have 8 operations defined. Read and write are by far the most commonly used ones; the rest are needed for virtual memory and operating system support. The branch function units have 12 operations defined which cover all condition code combinations used by VAX branches.

2.2.9 Distribution Bus

After a node is executed, its value must be distributed on the distribution bus to all locations that may depend on it. This includes the node tables, the RAT and the decoder. At each tag location, the tag on the distribution bus is compared to the stored tag. If they match, then the ready bit is set. At the same time, the value on the bus will be written into the value buffer.

Note that there is a distribution bus for each function unit. This does not cause contention problems, however, since each tag knows which distribution bus its result will appear on, and the tag comparisons only have to be done against that bus.

2.2.10 Checkpoint/Repair

The Checkpoint/Repair mechanism is used to ensure that precise interrupts will occur, and to maintain the sequential semantics of the VAX architecture. There are two major reasons for this mechanism: branch prediction and exceptions. In the case of an incorrect branch prediction, the effects of all instructions after the branch must be undone, and the correct execution path must be followed. For exceptions, the effects of all instructions after the one that caused the exception must be undone, and an exception handler must be executed. The simulator currently treats these situations identically.

When an exception or incorrect branch prediction is recognized, a repair is initiated. Repair involves restoring the state to a point in the instruction stream corresponding to the instruction that caused the problem, and invalidating all nodes merged after this instruction. Execution then resumes at the correct point, either an interrupt handler in the case of exceptions, or the correct branch direction in the case of branches.

3 Simulation Model

The simulator is written in about 24,000 lines of C. It consists of the microVAX subset of the VAX architecture, plus an additional 5 VAX instructions for a total of 193 VAX instructions implemented. There are 132 instructions unimplemented. The unimplemented instructions fall into three categories: floating point, decimal, and character string. Two character string instructions are implemented, MOVC3 and MOVC5; they are used for block transfers. The memory system is implemented as follows. Instruction fetches each take 1 cycle, assuming a 100% prefetch buffer hit rate. We also assume a 100% TLB hit rate and a 100% cache hit rate for all data accesses. Therefore, all data accesses take the same number of cycles, as specified by the memory function unit latency.

3.1 Simulator Flow Chart

The simulator uses a two phase clocking scheme. It assumes that all data structures (latches, registers, etc.) can be accessed twice per cycle, once in the first phase and once in the second phase.

The first phase includes the scheduling and merging stages of the pipeline. The scheduler must read each node table, presumably near the beginning, to send a node to the execution units. The merger must read and update the contents of the register and operand stack alias tables. After resolving data dependencies, the merger writes nodes to the rear of the node tables.

The checkpoint/repair mechanism also works during the first phase. Checkpointing the state of the alias tables is a simple operation that occurs in parallel with the merger alias table accesses. Repair is more involved. When a repair is initiated, no scheduling or merging is allowed to occur that cycle. The retirement mechanism also occurs during the first phase as it is closely related to the checkpoint/repair mechanism.

The second phase includes decoding and distribution. The decoder generates nodes for the merger, but does not otherwise interact with the main datapath. The distribution stage will

benchE	string search
benchF	linked list insertion
benchH	bit test and set
hanoi3	Towers of Hanoi with 3 disks
linpack	the daxpy subroutine from linpack
dhrynr	Dhruystone, with no registers

Figure 2. Benchmarks

update the contents of the node tables and alias tables, and hence must occur in the opposite phase from both the merging and scheduling phases.

The execution units in this scheme take as long as necessary. Operations which are assumed to execute in a single cycle will be scheduled and distributed in the same cycle.

3.2 Benchmarks

The HPSVAX simulator takes as input a UNIX executable program, and executes the program. The simulator provides a bare VAX execution environment. For example, virtual memory won't work unless you set up the page tables yourself. Also, system calls will not work unless you provide your own code for the system calls. As a result, the simulator can execute any compiled C program that does not have any system calls. This necessarily excludes I/O, but does not exclude most of the library routines. The benchmarks used for this paper are shown in Figure 2.

The initial set of benchmarks are all relatively small programs. As a result, they do not provide a good test of many HPS features, such as the node cache. Future work will include measuring the performance of larger benchmarks on the simulator.

BenchE is a string search benchmark. This program spends most of its time in a simple 4 instruction loop. Since this loop can be predicted well, this benchmark is near 1 cycle/instruction for most simulator configurations. Also, six VAX instructions account for 86% of all instructions executed.

BenchF is a bit test and set benchmark. In this benchmark 12 instructions account for 90% of those executed.

BenchH is a linked list insertion benchmark. Two instructions, MOVL and CLRL, comprise 44% of those executed.

The Hanoi3 benchmark is a recursive program that solves the Towers of Hanoi problem with three disks. Six instructions account for more than 98% of those executed. Also, almost one fourth of the instructions executed are CALLS or RET instructions. This is far from the norm for C language programs.

The linpack benchmark is the daxpy subroutine from the linpack benchmark set. This in an integer precision version of the subroutine. Three instructions account for more than 98% of those executed.

The Dhrystone benchmark is a C version of the ADA benchmark written by Reinhold P. Weicker. This benchmark was developed by examining the dynamic instruction set usage of typical programs, and then written to try to match those statistics. Eighteen instructions account for 90% of those executed.

3.3 Parameters

Figure 3 shows a sample configuration file for the simulator. The window size indicates how many instructions are allowed to be active at a time. When there are this many instructions in the active window, the decoder will stall until the oldest one retires. The larger the window, the more entries are needed for the node tables. In this case, the node tables have 64 entries each. The value buffer has 64 entries in each of its four banks: LIT, FIX, MEM, and REG. Note that the branch unit doesn't have a bank because it does not produce results.

The merger bandwidth is assumed to be the same as the number of functions units, except for memory nodes and literals. There is never more than one memory function unit, i.e. the memory system is assumed to be able to initiate only one access at a time. The literals do not have an associated function unit. They are used to insert 32 bit quantities into the datapath (such as static registers, absolute addresses, etc.) and are directly distributed after being merged. The floating point function units are not implemented since the microVAX subset does not include floating point instructions. We only have one branch unit since, in the absence of string instruction optimization, VAX instructions contain at most one branch node each.

For each instruction, the register/memory write buffer indicates which register/memory locations are modified by the instruction. This information is used by the retirement system to move register values to the permanent area of the VB, and to make memory writes permanent by sending them to the main memory. The Memory alias table is an associative buffer that holds all memory writes until retirement time.

Two parameters, the register and memory write retirement rates, indicate how many writes, of each type, can be retired per cycle. We are developing schemes to make retirement transparent to the rest of the machine. This is reflected in the large default values for retirement rates that we use in the simulator.

The final group of parameters specify the node cache. The node cache size indicates the number of entries, where each entry consists of all of the nodes for an instruction. The set size indicates the associativity of the node cache. The Hit Decode Penalty indicates how many cycles it takes the decoder to generate nodes for an instruction when there is a node cache hit. The Miss Decode Penalty indicates the number of cycles for node generation when there is a node cache miss.

Not shown in the figure are the latency figures for the function units. The Fixed Point and Branch function units are usually assumed to take one cycle. The memory unit default latency is two cycles.

The values shown in Figure 3 comprise the default configuration for all the simulations below. When a set of parameters are varied, the rest of the parameters are assumed to be these default values.

```
8       Window Size
64      Fix Point Node Table Bank Size
64      Memory Node Table Bank Size
64      Branch Node Table Bank Size
64      VB Bank Size
4       Number of Fixed point FU
2       Memory Merging Bandwidth
0       Number of Floating Point FU
1       Number of Branch FU
4       Literal Merging Rate
64      Register Write Buffer Size
64      Memory Write Buffer Size
64      Memory Alias Table Size
1024    Node Cache Size
4       Node Cache Set Size
1       Node Cache Miss Decode Penalty
1       Node Cache Hit Decode Penalty
16      Register Write Retire Rate
16      Memory Write Retire Rate
```

Figure 3. Standard Configuration File

4 Measurements and Analysis

This section describes the tuning experiments that we have performed with the simulator. Many parameters in the simulator can be varied. We have chosen to tune the window size, number of function units, memory latency, function unit latency, and decode delay, because of the likelihood of these parameters to influence performance.

4.1 Window Size

The active window size strongly dictates the amount of hardware in the machine. As the window size increases, the memory alias table, all of the node tables, and also the value buffer must increase in size to hold the extra nodes. The window size also has an effect on the parallelism that can be extracted from the instruction stream. The larger the window size, the more nodes are in the machine and the greater the chance that local parallelism can be exploited.

Figure 4 shows the effects of changing the window size. As expected, performance increases as window size increases. Note that for the benchmarks tested, there is no gain from increasing the window size from 12 to 16. There is however a large increase in going from 4 to 8, so the window size should be at least 8. Since a window size of 8 yields most of the performance possible, for these benchmarks, it is a good benefits/cost choice for this implementation. The hardware needed is a node table of size 64 and a value buffer of 256 entries. Both of these values are reasonable for hardware implementations.

4.2 Number of Function Units

The number of fixed point function units is varied also. The fixed point units are assumed to be all identical. Much of the work is done by these function units, such as address generation, register increment/decrement, etc. Therefore it is useful to have as many as possible. However, multiple function units require more logic in terms of the function units themselves, additional distribution buses, and, if we wish to effectively load balance the multiple function units, additional scheduling hardware. This would argue for keeping the number of function units as low as possible.

Figure 5 shows the effects of multiple fixed point function units upon performance. There is a large performance increase from 1 to 2 units, a moderate increase from 2 to 4, and very little increase afterwards. The flatness of the graph after four function units is due in large part to the fact that our implementation merges one instruction per cycle. When so limited, there is not enough work to keep 8 function units busy. Also, since we assumed a naive method of scheduling fixed point nodes, which makes no attempt to balance the load, we can not effectively utilize the additional function units. With a substantial increase in decoding, merging, and scheduling hardware, we could obtain improved results for large numbers of function units. For this implementation however, four fixed point function units offers a good balance between performance and cost.

4.3 Memory Latency

Memory latency is another important parameter affecting performance. (See Figure 6.) HPS, with its out of order execution, should not be drastically affected by memory system delays, provided that memory contention is not caused by multiple active memory nodes in various stages of execution. In fact, we have assumed that memory accesses are perfectly pipelined, so that one memory access can be started every cycle no matter what the latency is. With this assumption, the performance of HPS shows very little effect from the increase in the memory latency. This demonstrates that HPS can effectively overlap operations, and execute operations out of order to reduce the effect of the memory latency, given this caveat about contention. We should also point out that although the concept of perfect pipelining of the memory accesses may appear at first to be unrealistic, Swensen[9] has suggested a circuit for facilitating pipelined memory accesses. Also, a sufficient degree of interleaving in the main memory would effectively produce the benefits of pipelining. How much inter-

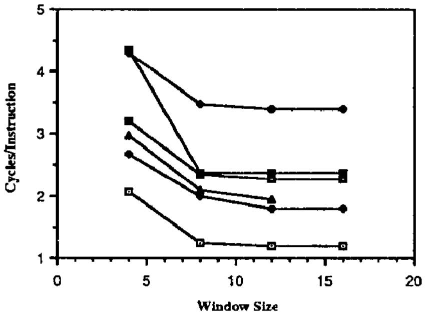


Figure 4. EFFECT OF WINDOW SIZE

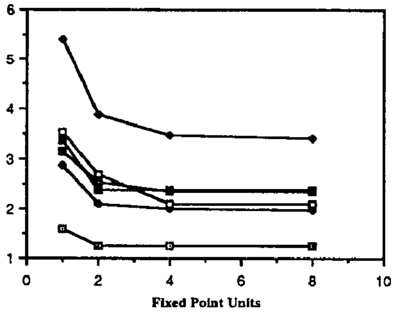


Figure 5. EFFECT OF NUMBER OF FIXED POINT UNITS

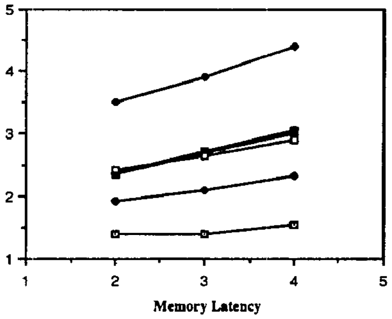
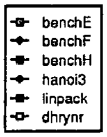


Figure 6. EFFECT OF MEMORY LATENCY



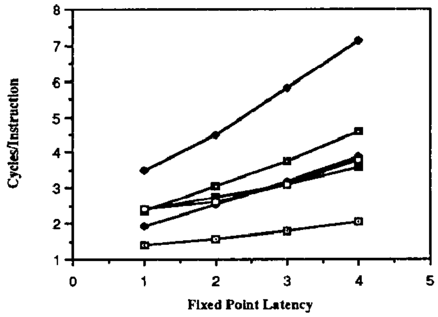


Figure 7. EFFECT OF FIXED POINT LATENCY

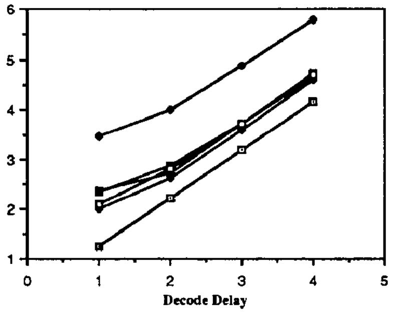


Figure 8. EFFECT OF DECODE DELAY, NO NODE CACHE

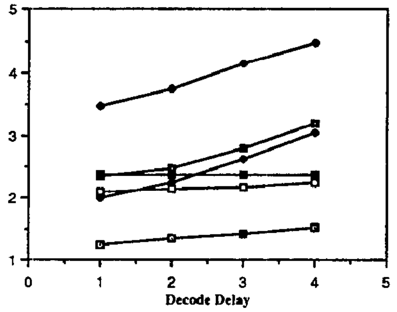


Figure 9. EFECT OF DECODE DELAY WITH NODE CACHE

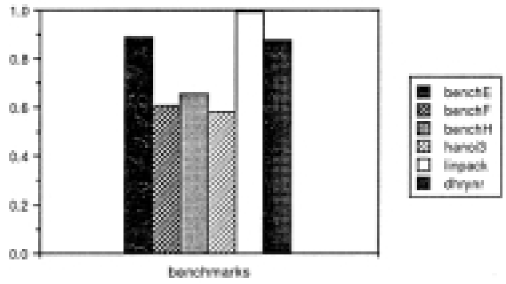


Figure 10. NODE CACHE HIT RATE

leaving would be required to accommodate this is still an open question.

4.4 Function Unit Latency

Fixed point function unit latency is also an issue. Again, the function units are assumed to be perfectly pipelined so that each fixed point function unit can start one alu operation every cycle. This is not at all difficult to implement, and in fact can be a big win since it potentially allows dramatic reduction of the cycle time. Figure 7 shows the effects of changing the function unit latency. For four of the benchmarks, the performance decreases by roughly one half when the fixed point function unit latency is increased from 1 to 4 cycles. For the other two, the decrease is less than one half. This is due to the fact that HPS is able to effectively exploit concurrency by overlapping operations, thereby reducing the performance penalty due to the increase in the function unit latency.

4.5 Node Cache

Since the decoder is one of the most complicated parts of a VAX implementation, it is important to know the effects of decode delays on performance. The node cache handles much of this problem by caching instructions in decoded form. Our initial benchmarks are small enough to fit into a reasonably sized node cache. Hence the node cache hit rate is dominated by the percent of the instructions executed that are actually cacheable. This will change when we expand our set of benchmarks.

Figure 8 shows the effect of varying the decode delay from 1 to 4 cycles in the absence of a node cache. The decode delay is defined here as the number of cycles until the first node is generated, with all nodes then being available at the same time. With a decode delay of four cycles, the best possible performance is four cycles per instruction. So, predictably, the performance decreases as the decode delays increase. Note also, that as the decode delays increase, the performance tends to get closer to the theoretical maximum, as the HPS microengine is better able to overlap the decode and execute stages of each instruction.

Figure 9 shows the effect of varying the decode delay when a node cache is present. We note that most of the benchmarks show very little effect on performance as most of their instructions can be cached. The hanoi benchmark, for example, has a noticeable performance decrease. This is because hanoi executes many CALLS and RET instructions which can not be cached.

Figure 10 shows the node cache hit rates for the benchmarks. Linpack has a 99% hit rate which is why it is very insensitive to the decode delays in the presence of a node cache. The worst is hanoi3, at 58%. As mentioned above, the node cache at present is overly pessimistic in deciding when it can cache an instruction, so we expect to be able to improve on these results.

5 Conclusions

HPS offers the potential of a high performance implementation for traditional architectures, by providing high bandwidth, few stalls, multiple function units and out of order execution. It is relatively insensitive to parameters such as memory latency due to the out of order execution of nodes. One of the more complicated parts of a VAX implementation, the decoder, has only a moderate effect on the performance as many instructions can be cached in their decoded form (assuming that the node cache is large enough to get a good hit rate.)

The results suggest that a reasonable implementation, as defined by the standard configuration file, could execute at the rate of 2 cycles per VAX instruction. This compares very favorably with existing implementations which all take over 6 cycles per instruction. We must be quick to point out, however, that this simple comparison does not take into account some of the real world problems of computers (cache misses, TLB misses, etc.), and also suffers from the absence of real world substantial benchmarks. On the other hand, it does not fully exploit the capabilities of HPS (unnecessary decoder stalls, etc.). In summary, this paper shows that there are performance benefits to be gained by using the HPS concepts, and that future refinements are warranted.

6 Acknowledgements

The authors wish to acknowledge the Digital Equipment Corporation and the NCR Corporation for their generous support of out research. We also would like to acknowledge two other members of our research group, Chien Chen and Jiajuin Wei, who have made contributions to this project. Part of this work was sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract No. N00039-84-C-0089. Finally, we also acknowledge that part of this work was supported under a NSF Graduate Fellowship.

References

[1] Anderson, D. W., F. J. Sparacio, and R. M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction Handling," *IBM J. of R & D*, vol. 11, no. 1, 1967.

[2] Arvind and K. P. Gostelow, "A New Interpreter for Dataflow and Its Implications for Computer Architecture," Department of Information and Computer Science, University of California, Irvine, Tech Report 72, October 1975.

[3] Dennis, J. B., and D. P. Misunas, "A Preliminary Architecture for a Basic Data Flow Processor," *Proceedings of the Second International Symposium on Computer Architecture*, 1975, pp. 126-132.

[4] Hwu, W., S. Melvin, M.C. Shebanow, C. Chen, J. Wei, and Y.N.Patt, "An HPS Implementation of the VAX; Initial Design and Analysis," *Proceedings of the 19th Annual Hawaii International Conference on System Sciences*, 1986.

[5] Hwu, Wen-mei, and Yale N. Patt, "Checkpoint Repair for High Performance Out-of-order Execution Machines," *Proc. 14th Int. Symp. on Comp. Arch.*, June 2-5, 1987, Pittsburgh, Pa.

[6] Patt, Y.N., Wen-mei Hwu, Mike Shebanow, "HPS, a New Microarchitecture: Rationale and Introduction," *Proc. of the 18th Microprogramming Workshop*, Asilomar, California, Dec., 1985.

[7] Patt, Y. N., S. Melvin, W. Hwu, M. C. Shebanow, C. Chen, J. Wei, "Run-Time Generation of HPS Microinstructions From a VAX Instruction Stream," *Proceedings to the 19th Annual Workshop on Microprogramming*, October 15-17, 1986, New York, New York.

[8] Tomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J. of R & D*, vol. 11, no. 1967, pp 25 - 33.

[9] Swensen, John, "High-Bandwidth/Low-Latency Temporary Storage for Supercomputers," *PhD Thesis*, U.C. Berkeley, Nov., 1987.