Performance Benefits of Large Execution Atomic Units in Dynamically Scheduled Machines

Stephen W. Melvin

Computer Science Division
University of California
Berkeley, CA 94720

Yale N. Patt

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2110

ABSTRACT

In this paper we identify three types of atomic units, or indivisible units of work: architectural atomic units (defined by architecture level interrupts and exceptions), compiler atomic units (defined by compiler code generation) and execution atomic units (defined by run-time interruptibility). We discuss tradeoffs for these units and show that size has different performance implications depending on the atomic unit. We simulate a number of different implementations of the VAX architecture, focusing on different execution atomic unit sizes. We show that significant performance benefits can be achieved by having large execution atomic units in dynamically scheduled machines.

1. Introduction

In this paper we investigate the notion of an indivisible unit of work, or an 'atomic unit.' There are different types of atomic units depending on with respect to what the work is atomic. We define three such types: architectural atomic units, compiler atomic units and execution atomic units.

We explain the subtleties of the distinctions between the three types of atomic units in section three. For now, it is sufficient to know that atomic units consist of 'nodes,' each of which represents a basic unit of work that a 'function unit' in the data path of a machine is capable of performing. By a basic unit of work, we mean a primitive arithmetic operation or a memory operation. By a 'function unit' we mean the hardware which executes a node, that is, an arithmetic unit or a path to memory. The 'size' of an atomic unit is the number of nodes that it encompasses.

Nodes are combined into 'multinodewords' in two places in the machine; as the set of nodes which are issued in the same cycle (an 'issue-multinodeword'), and as the set of nodes which are scheduled for execution in the same cycle (a 'schedule-multinodeword'). We distinguish between issuing and scheduling because in a dynamically scheduled machine they are distinct operations. Each node that has been issued waits in a 'node table', decoupled from the other nodes in the same issue-multinodeword, until all its source operands are ready and an appropriate function unit is available. The node table (which R. M. Tomasulo calls a reservation station [8]) in conjunction with a register alias table implement out-of-order scheduling.

If all function unit data paths are not fully pipelined, then the presence of a node in a particular schedule-multinodeword will prevent certain nodes from appearing in subsequent schedule-multinodewords. In a fully pipelined machine, however, there are no hardware restrictions on nodes in consecutive schedule-multinodewords.

In a previous paper [4], we presented the atomic unit framework and focused on a particular implementation technique for run-time generation of multinodewords. In this paper we focus on how the overall performance varies with different sizes of execution atomic units.

We have simulated several different dynamically scheduled implementations of the VAX architecture and measured their relative performance. These implementations vary in their function unit latency, their sizes of

the out-of-order execution 'window' and their sizes of execution atomic units. We show that significant performance benefits can be achieved by having large execution atomic units in dynamically scheduled machines.

This paper is divided into five sections. Section two presents the abstract machine model on which our experiments were based. Section three outlines our atomic unit framework, discusses atomic unit size tradeoffs and illustrates how several existing machines fit within the framework. Section four describes our experiment and analyzes our results. Section five offers some concluding remarks.

2. Abstract Machine Model

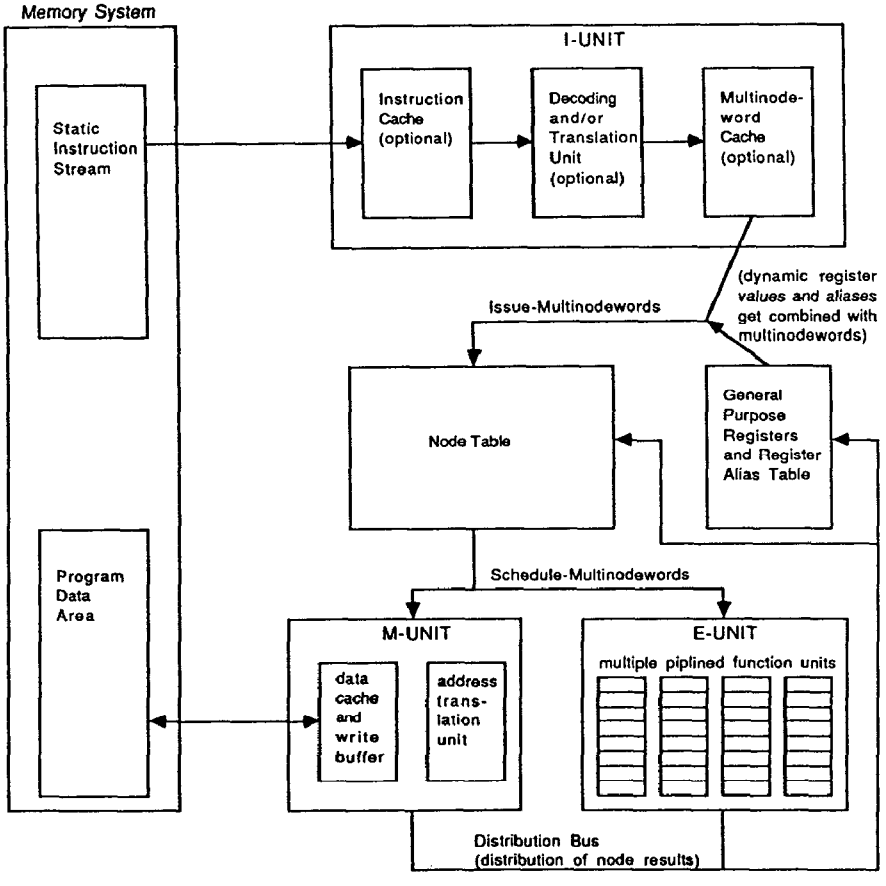
Our experiments are based on the abstract machine model illustrated in Figure 1. The intent of our model is to capture the essential elements of the differences we are interested in measuring without being overly tied to a specific implementation.

The instruction stream is fetched from memory and issued as multinodewords into the node table by the I-UNIT. If the instruction stream is already formatted into multinodewords, then it is passed directly to the node table. If not, it is first decoded (i.e., translated) into multinodewords before being issued. Nodes can be executed in an arbitrary order but older nodes take priority over younger ones.

Results are distributed from the function units both to the node table where other nodes may be awaiting these results as source operands and to the architecturally visible registers. We assume that the architectural registers are explicitly named in the static instruction stream and register synchronization is accomplished with the Tomasulo algorithm [8]. This tag forwarding mechanism, implemented by the register alias table, providing a logical (i.e. instruction set architecture specific) to physical (i.e., actual machine resource) mapping.

Our abstract machine model does not specify the control of the data path; that is, how the multinodewords are sequenced. We are concerned in this paper with the utilization of the data path, not the implementation of the control path. None the less, a few words are in order.

Figure 1
Abstract Machine Model



A program consists of sets of operations, or 'basic blocks,' which are sequenced in such a way as to implement the program being executed. The operations within a basic block are encoded as nodes within one or more issue-multinodewords and they are delimited by control instructions, identifying changes in the flow of control.

The I-UNIT interprets these control instructions and sequences the issue-multinodewords accordingly, using one of two mechanisms. Control flow changes may be predicted, allowing for the possibility that work will have to be undone, or the machine may be stalled when a branch is detected until the condition being tested has been generated. We have not specified the action of a control flow change as a node, but there may be certain related operations which are nodes (e.g. a subtract node for a branch on the result of a comparison).

3. Atomic Unit Framework

The abstract machine model described in the previous section is used as a basis for discussion in the remainder of this paper. In this section we present the atomic unit framework. We have identified three separate types of atomic units: *architectural atomic units (AAUs), compiler atomic units (CAUs),* and *execution atomic units (EAUs).* An architectural atomic unit is a group of nodes which is atomic with respect to architecturally defined interrupts and exceptions. That is, events which can interrupt the normal execution of a stream of code (hardware interrupts and process exceptions) and cause a change in the flow of control (typically to operating system code), take place in between architectural atomic units.

Thus, the state of a process can only be preserved at an AAU boundary. An AAU is normally what is thought of as a macro instruction since most processors are interruptible at this level. In principle, AAUs could encompas varying amounts of work, from individual nodes to entire basic blocks (or larger), but where they are significantly affects the way the hardware is implemented.

Small AAUs allow low interrupt latency to be easily handled in machines with simple data paths since the level of granularity is small. However, with suitable backup hardware interrupt latency can be minimized even with large AAUs by discarding pending operations when an interrupt is received. This sort of backup hardware is generally required in out-of-order execution machines and those with dynamic branch prediction, so it might not pose a significant increase in complexity, but note that due to the discarded work, efficiency is reduced.

Large AAUs potentially decrease the size of the architectural state of the machine. This is because within an uninterruptible block of nodes, temporary results need not be given explicit names, they can be referred to by their position within the block. That is, since all of the work within the AAU is atomic with respect to process state, values which are computed and used only within the block need not go through named registers. Thus, fewer architecturally defined general purpose registers may be required, making context switching less expensive.

The second type of atomic unit, the compiler atomic unit, is the smallest unit of work that the compiler is able to specify. As with AAUs, a CAU is normally what is thought of as a macro instruction, since the compiler generates code to that level. AAUs need not be the same as CAUs, however, and in principle they could be larger or smaller. In the case of AAUs larger than CAUs, the compiler would group together a set of individually produced units of work and define them to be uninterruptible. This would be useful in cases where the compiler generates individual nodes and architecture level interruptibility is not needed at that low a level. In the case of AAUs smaller than CAUs, the hardware can suspend execution in the middle of a compiler generated unit of work (e.g. a long string move instruction that page faults).

There are two main choices for CAUs: individual nodes and higher level encoded instructions encompassing from one to many nodes. If CAUs are individual nodes, then the set produced by the compiler could either be the exact set implemented by the hardware, or there could be some sort of translation which occurs at run-time. If the CAUs are encoded instructions, more than just a translation is involved; a decoding process is necessary in which nodes are generated. These nodes can either then be directly executed or they can be packaged into larger units and executed together. For either type of CAU, it may be useful to cache the nodes in order to save the work of translating or decoding.

An important issue which relates to the format of the node type CAUs is the notion of the degree of 'coupling' to the implementation. The nodes produced by the compiler may be freely formated, in which case they are decoupled from the implementation, or alternatively they may be formated exactly into the issue-multinodewords of a specific implementation. In the former case, the nodes that constitute the instruction stream must be packed into issue-multinodewords dynamically. This is accomplished through the use of what we refer to as a 'fill unit' [4]. Although this scheme requires greater run-time overhead, an advantage is that a buffer against technology is provided. Machines with varying numbers of nodes per multinodeword could be implemented in a straightforward way. Also, a machine with uncoupled CAUs can take advantage of run-time information (such as branch prediction information or dynamic operand values) in creating multinodewords.

Small compiler atomic units may increase compile-time effort but allow more compile-time optimization by allowing a finer level of control over the work to be done. The definitions of the architectural atomic units and the compiler atomic units together we collectively refer to as the 'instruction set architecture' (ISA). Conventional ISAs have only a single set of instructions, which act as both AAUs and CAUs.

The third and final type of atomic unit, the execution atomic unit, is an implementation issue rather than an architectural one. An EAU is a group of microoperations that the hardware issues at run-time as an indivisible unit. The inability to partially execute an EAU stems from the forwarding of node results to other nodes without going through explicit registers. Thus, an EAU is analogous to an AAU, but rather than a constraint imposed by the architecture, an EAU is a characteristic of an implementation.

An EAU may be smaller than an AAU (for example in a sequential microcoded machine where each node gets executed individually) or it may be larger (in cases where the hardware dynamically creates large units of indivisible work). Larger EAUs are more likely to lead to higher utilization of node slots within the multinodewords since the work constituting an EAU is packed into issue-multinodewords as a unit.

In the case of machines with CAUs equal to nodes and closely coupled to an implementation, the compiler has the opportunity to pack the nodes exactly into issue-multinodewords, as mentioned above. In this case, the compiler can use static branch prediction, loop unrolling and global optimization techniques, such as trace scheduling [1], to get higher utilization. Having EAUs larger than multinodewords will not increase utilization in this case since the compiler is in control of the exact multinodeword composition. This is true for statically as well as dynamically scheduled machines, as long as the instruction unit doesn't do any translation or repacking of multinodewords.

On the other hand, machines which have encoded CAUs and machines with node type CAUs uncoupled from the implementation may benefit from multiple multinodeword EAUs. In this case, the instruction unit will dynamically package nodes into EAUs. If the EAUs are larger, there is a greater chance that more node slots can be utilized since the larger set of nodes to choose from will tend to smooth out constraints on which types of nodes can be placed in which locations within a multinodeword.

One advantage of large AAUs not mentioned above is that they make it easier to increase the size of EAUs. Implementing EAUs larger than AAUs is non-trivial and requires the implementation of some sort of backup hardware [2]. In the event of an exception in the middle of an EAU (but at the boundary of an AAU), the backup hardware restores the state of a machine to the beginning of the EAU Execution then proceeds one AAU at a time until the exception is discovered. We define the operation of a machine implementing this type of backup hardware as having two modes: fast and slow. In fast mode, the machine executes EAUs and in slow mode the machine executes AAUs (which are smaller).

In the IBM 801, the justification was made for small compiler atomic units in order to exploit their advantages (i.e. a finer level of control over the nodes which are executed) [7]. Other processors available today have also used this philosophy. These architectures generally have AAUs the same as the CAUs, and the idea that in any implementation, the EAUs will also be the same. Thus, while these architectures can benefit from the performance advantages of small CAUs, they are limited in that implementing them on hardware with large EAUs is difficult.

That is, if an architecture similar to the IBM 801 were to be implemented on a processor with a high degree of concurrency, high function unit utilization would necessitate large EAUs (at least as large as multinodewords). While this could be accomplished by implementing a fast mode / slow mode scheme and dynamically creating multinodewords, it would be more natural to discard the architecture and make a transition to a VLIW machine, where the AAUs and EAUs increase in size to a multinodeword but CAUs remain at the node level. In this case, the nodes are scheduled statically, so increasing the size of the EAUs beyond the size of the multinodewords won't increase utilization. The compiler has already packed the node slots in the multinodeword.

To further increase utilization, implementation techniques such as HPS dynamically schedule nodes onto function units [5]. In an HPS implementation that has CAUs closely coupled to the implementation, such as HPSm [3], EAUs and AAUs are again equal to multinodewords. An HPS machine with EAUs encompassing multiple multinodewords can further increase utilization. By allowing a larger unit of work to be put together as an atomic unit, more node slots can potentially be filled. In addition, by utilizing dynamic information in creating EAUs, even further run-time optimization is possible. Finally, dynamic branch prediction in a machine of this type would potentially allow EAUs to encompass more than one basic block, further increasing function unit utilization. That is, two consecutive basic blocks joined by a highly predictable branch could be joined together into a single EAU. In this case, however, we would be back to a fast mode / slow mode scheme unless AAUs are also increased beyond the size of a basic block.

4. Experimental Results

In this section we describe the measurements we made on the performance of dynamically scheduled machines with different sizes of execution atomic units. We measured 9 different implementations for each of the two EAU possibilities: VAX instructions and basic blocks. There were two parameters which we varied, window size and function unit latency. The compiler atomic units were VAX instructions in all cases.

The window size, which is the amount of active work permitted in the machine was simulated at one EAU, two EAUs and an unlimited number. This permits us to see how different choices of EAU sizes varies as a function of how much work is active. Latencies of ALU/memory were simulated at cycle counts of 1/1, 4/4 and 4/8. This permits us to see how EAU size interacts with the depth of the function unit pipelines.

Figure 2
Benchmark Summary

1. String Search	
Number of VAX Instructions:	707
Number of Basic Blocks:	223
Average Number of Nodes per VAX Instruction:	1.59
Average Number of Nodes per Basic Block:	5.03
2. Linked List Insertion	
Number of VAX Instructions:	331
Number of Basic Blocks:	64
Average Number of Nodes per VAX Instruction:	2.34
Average Number of Nodes per Basic Block:	12.11
3. Livermore Loop #1 (Hydro Excerpt)	
Number of VAX Instructions:	1914
Number of Basic Blocks:	104
Average Number of Nodes per VAX Instruction:	1.33
Average Number of Nodes per Basic Block:	24.45

We made the 18 measurements described above on three benchmarks: a string search benchmark, a linked list insertion benchmark and a Livermore Loop kernel. The benchmarks are summarized in figure 2 and the results are presented in figure 3. We indicate the total number of cycles for each implementation, the performance relative to the unlimited window, low latency, basic block EAU case and the function unit utilization. Note that since some work gets scheduled and then discarded, the utilization numbers are not directly related to the total execution time.

We have focused on issue bandwidth and data path utilization, not node generation, so we do not examine here the run-time overhead that an encoded CAU machine would have over a node type CAU machine. We only note that it should be a second order effect if an effective node cache is implemented. This is because the work of generating nodes only occurs on a node cache miss.

The simulator generates nodes according to the instruction stream and then fills them into multinodewords, adjusting all intra-instruction and intra-EAU node references. Multiple writes to the same register within an EAU are removed, so that at most one value is written into any register. In the current version, we define multinodewords to contain one memory node and three ALU nodes (ALU nodes can be placed anywhere within the three node slots of a multinodeword).

The EAUs, once filled, are passed on to a merge unit, where they are issued to the node table one multinodeword per cycle. This is where static to dynamic binding occurs. Each node that is issued is assigned a global dynamic tag, and reads from registers and writes to registers are processed by the register alias table. In each cycle, nodes are scheduled from the node table into each function unit. We currently implement one function unit per multinodeword slot, thus three ALU nodes and one memory node can be scheduled per cycle.

Every cycle results being distributed from the function units or the memory unit are used to update waiting nodes in the node table and stored in the register alias table for future use. The node tables and register alias table are divided into banks, which correspond to EAUs. When a branch prediction trap occurs (signaled by an "assert" node which gets executed by function

Figure 3
Benchmark Results

	Basic Block EAUs				VAX Instruction EAUs			
	# of cycles	- ALU util.	- MEM util.		# of cycles	- ALU util.	- MEM util.	
1. String Search								
Unlimited Window - Latency 1/1	505	1.00	0.53	0.06	828	0.61	0.31	0.04
Unlimited Window - Latency 4/4	946	0.53	0.33	0.03	1053	0.48	0.26	0.03
Unlimited Window - Latency 4/8	998	0.51	0.32	0.03	1023	0.49	0.27	0.03
Two EAU Window --- Latency 1/1	1216	0.42	0.20	0.02	2240	0.23	0.11	0.01
Two EAU Window --- Latency 4/4	2275	0.22	0.11	0.01	4206	0.12	0.17	0.01
Two EAU Window --- Latency 4/8	2742	0.18	0.09	0.01	5130	0.10	0.14	0.01
One EAU Window --- Latency 1/1	2281	0.22	0.11	0.01	2994	0.17	0.08	0.01
One EAU Window --- Latency 4/4	4298	0.12	0.17	0.01	5160	0.10	0.14	0.01
One EAU Window --- Latency 4/8	5179	0.10	0.05	0.01	6128	0.08	0.04	0.01
2. Linked List Insertion								
Unlimited Window - Latency 1/1	582	1.00	0.31	0.30	628	0.93	0.84	0.29
Unlimited Window - Latency 4/4	993	0.59	0.19	0.18	1046	0.56	0.17	0.17
Unlimited Window - Latency 4/8	1190	0.49	0.16	0.15	1245	0.47	0.15	0.15
Two EAU Window --- Latency 1/1	653	0.89	0.25	0.24	1073	0.54	0.15	0.14
Two EAU Window --- Latency 4/4	1123	0.52	0.15	0.14	1955	0.30	0.08	0.08
Two EAU Window --- Latency 4/8	1339	0.43	0.13	0.12	2399	0.24	0.07	0.06
One EAU Window --- Latency 1/1	881	0.66	0.18	0.18	1598	0.36	0.10	0.10
One EAU Window --- Latency 4/4	1534	0.38	0.10	0.10	2848	0.20	0.06	0.05
One EAU Window --- Latency 4/8	1826	0.31	0.09	0.08	3468	0.17	0.05	0.04
3. Livermore Loop #1 (Hydro Excerpt)								
Unlimited Window - Latency 1/1	935	1.00	0.58	0.22	1845	0.51	0.29	0.11
Unlimited Window - Latency 4/4	957	0.98	0.57	0.22	1885	0.50	0.28	0.11
Unlimited Window - Latency 4/8	963	0.97	0.57	0.22	1900	0.49	0.28	0.11
Two EAU Window --- Latency 1/1	1285	0.73	0.42	0.16	4746	0.20	0.11	0.04
Two EAU Window --- Latency 4/4	2505	0.37	0.22	0.08	8977	0.10	0.06	0.02
Two EAU Window --- Latency 4/8	2913	0.32	0.56	0.07	11389	0.08	0.05	0.02
One EAU Window --- Latency 1/1	2442	0.38	0.22	0.09	8557	0.11	0.06	0.02
One EAU Window --- Latency 4/4	4869	0.19	0.11	0.04	15794	0.06	0.03	0.01
One EAU Window --- Latency 4/8	5681	0.16	0.10	0.04	19406	0.05	0.03	0.01

unit), the sequencer in the I-UNIT backs up the node tables and the register alias table and proceeds to merge the EAU at the alternate direction. A two-bit counter branch predictor is implemented.

When all registers are valid in the register alias table and all nodes in the node table have been scheduled for the oldest bank in the machine, it is retired. This causes all memory writes to be made permanent, the literal register values are copied into a permanent area and the bank is usable for the next EAU. We don't count the cycles that would be needed to generate multinodewords from encoded VAX instructions. That is, we assume that the decoded EAUs can always be pulled directly from a node cache.

The main limitation of the VAX instruction EAU implementations is that instructions have to be split on multinodeword boundaries. Thus, if an instruction only consists of one node, three slots will be unused. Also, if an instruction requiring two memory nodes and no ALU nodes is followed by an instruction requiring six ALU nodes, a total of four multinodewords are required instead of the minimal two. In the basic block EAU implementation, consecutive instructions can share multinodewords arbitrarily.

Summaries of the benchmarks are shown in figure 2. Note that CAU size ranges from 1.59 nodes to 2.34 nodes. We see that the sizes of the basic blocks range from 5.03 in the string search benchmark to 24.45 in the Livermore Loop benchmark.

The results indicate the depending on the implementation, the speedup in increasing the size of the EAUs can be significant. In cases where issue bandwidth is a bottleneck, the speedup is naturally greater than in cases where other bottlenecks come into play. In the latency 4/8 cases, memory intensive benchmarks such as the string search are more likely to be limited by the memory rather than the issue bandwidth. Note that the difference between the basic block and VAX instruction implementations is low in the high latency cases for string search but greater for low latency implementations.

The Livermore Loop benchmark indicates different behavior. Because it is very compute dependent, even with high memory latency it is much more sensitive to issue bandwidth. In fact, the relative performance between basic block EAUs and VAX instruction EAUs is greater for high memory latency.

5. Conclusions

In a microarchitecture with multiple pipelined function units, the goal is to keep as many pipeline slots as possible full. This becomes more difficult as the number of pipeline slots increases. Each active pipeline slot represents an operation which is independent of all other operations active at that moment. Thus, in order to achieve high function unit utilization in machines with many pipeline slots, parallelism must be extracted from the code.

Out-of-order execution machines attempt to extract local parallelism from programs by scheduling nodes onto function units whenever their operands are ready, keeping the function units supplied with work to do. Unfortunately, function unit utilization can be low even when executing programs that have a large amount of local parallelism. One of the reasons for this is that nodes may not be optimally packed into multinodewords. The machine may stall due to lack of work, not because there is not enough work to do, but simply because it cannot be issued fast enough.

In order to investigate this problem, we have simulated several configurations with different sizes of execution atomic units. We found that in most implementations, a significant amount of additional local parallelism can be exploited when the execution atomic units are large. In a machine that can issue several nodes per cycle, the packing of nodes into multinodewords is a critical function. It may be done statically or dynamically, but larger units of work packed together lead to higher utilizations of node slots.

REFERENCES

[1] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, vol. C-30, no. 7, July 1981.

[2] Wen-mei W. Hwu and Yale N. Patt, "Checkpoint Repair for High Performance Out-of-order Execution Machines," *IEEE Transactions on Computers*, December 1987.

[3] Wen-mei W. Hwu and Yale N. Patt, "HPSm, A High Performance Restricted Data Flow Architecuture Having Minimal Functionality," *Proceedings, 13th Annual International Symposium on Computer Architecture*, Tokyo, June 1986.

[4] Stephen W. Melvin, Michael C. Shebanow and Yale N. Patt, "Hardware Support for Large Atomic Units in Dynamically Scheduled Machines," *Proceedings, 21st Annual Workshop on Microprogramming and Microarchitecture*, San Diego, CA, November 1988.

[5] Patt, Yale N., Hwu, Wen-mei, and Shebanow, Michael C., "HPS, a New Microarchitecture: Rationale and Introduction," *The 18th International Microprogramming Workshop*, Asilomar, CA, December 1985.

[6] Yale N. Patt, Michael C. Shebanow, Wen-mei Hwu and Stephen W. Melvin, "A C Compiler for HPS I, A Highly Parallel Execution Engine," *Proceedings, 19th Hawaii International Conference on System Sciences*, Honolulu, HI. January 1986.

[7] George Radin, "The 801 Minicomputer," *IBM Journal of Research and Development*, Vol. 27, No. 3, May 1983, pp. 237–246.

[8] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, Vol. 11, 1967, pp. 25–33.