Superscalar Execution With Dynamic Data Forwarding*

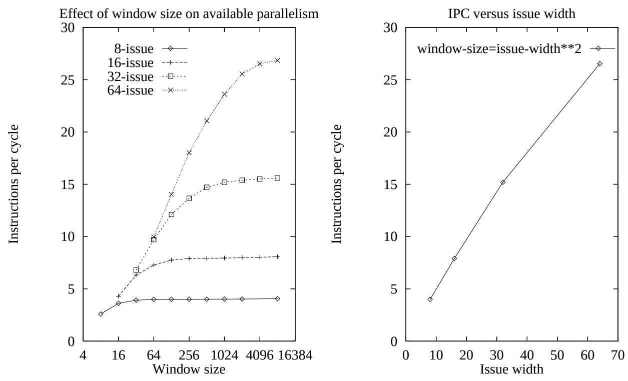
Soner Önder Rajiv Gupta
Department of Computer Science
University of Pittsburgh
Pittsburgh PA 15260
{soner,gupta}@cs.pitt.edu

Abstract

We empirically demonstrate that in order to take advantage of increasing issue widths, superscalar processors require quadratically growing instruction window sizes. Since conventional central window design aims to provide full data fan-out to all the instructions which are in the window, designing large instruction windows using conventional techniques is not feasible. We show that full data fan-out is not necessary for achieving high performance when a novel approach is used to distribute the values. We use direct matching using a small on chip memory called the wait memory *to implement the instruction window and bring in a small subset of instructions which are likely to become ready into a* match unit *where instruction selection and operand matching tasks are performed. We show that the match unit needs to grow only* linearly *with the issue width. We use SPEC95 benchmarks to demonstrate that at a given instruction window size our algorithm provides over 90 percent of the IPC that can be obtained by a central window implementation that provides full data fan-out.*
Keywords: *Wide-issue superscalar, instruction window, data fan-out, direct matching, dynamic forwarding.*

1. Introduction

Recent research indicates that issuing a large number of instructions per cycle may yield high performance, and processors that can issue large numbers of instructions will become feasible with the advances in manufacturing techniques [3, 7]. Some recent research has focused on the problems of the processor front-end such as equipping the fetch unit with accurate branch predictors which can perform multiple branch predictions and increasing the fetch bandwidth without increasing the fetch latency [2, 8]. Research in this area of the processor design has been yielding increasingly promising results.



(a) (b)

Figure 1. SPEC95 Behavior

Unfortunately, even with a perfect instruction fetcher, the processor core still needs improvement. Providing large issue widths alone is not sufficient for exploiting greater degrees of instruction level parallelism. With larger issue widths, instruction window must also be enlarged. Fig. 1(a) indicates the performance of processors with a perfect front-end and perfect memory disambiguation. As it can easily be seen, the knee for these curves is a quadratic function of the issue width. When the window size is enlarged quadratically with increasing issue width almost a linear speed-up is obtained (Fig. 1(b)). Designing such large windows is not feasible with the conventional approaches, which provide full data fan-out by using a *broadcast and select* mechanism [9] due to the delays involved in the wake-up logic, long wires used to broadcast the tags (and data) to the waiting instructions, and large number of comparators needed to implement the instruction window [5].

*This work is supported by NSF grants CCR-9704350, CCR-9808590 and a grant from Intel Corporation.

To address these problems, various techniques have been proposed. These techniques attempt to partition the central window either in the form of a clustered architecture where the machine has multiple instruction windows [5], or in the form of trace lines [10] as well as using dependency information to reduce the number of instructions being considered for selection. In this work, we take these approaches one step further by handling the problem as a data-flow graph problem. We illustrate that for high performance the full data fan-out that is provided by the central window approach is not necessary when a new technique is employed to distribute the data values. Our approach relies on a graph which is computed from a conventional instruction set dynamically and is called the *Dynamic Data Forwarding Graph* (DDFG). The semantics of DDFG can easily be implemented using *direct matching* and unlike the prior direct matching techniques [11, 6], the graph does not limit parallelism or introduce additional instructions to carry out data propagation. The graph is based on the novel idea of using instructions which are being woken-up as stepping stones to waking-up further instructions. We further show that use of a DDFG provides competitive performance to that of a fully-associative central instruction window and naturally leads to clustering.

In the remainder of the paper, in section 2 we present our novel idea of dynamic data forwarding. In section 3, we present a preliminary implementation of our algorithm and discuss its hardware complexity. Finally, in section 4, we discuss the results of a comparative study of dynamic data forwarding architecture with a conventional superscalar processor.

2. Dynamic Data Forwarding

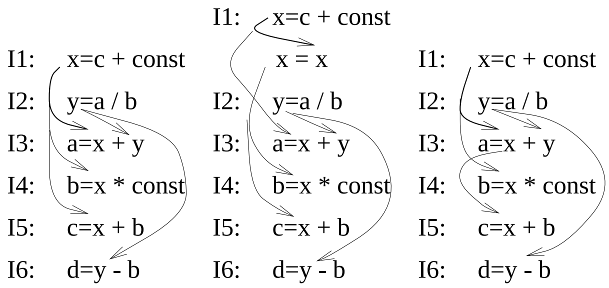
Although direct matching has been studied extensively in the context of data-flow architectures by Papadopoulos and Culler [6], the technique has never been applied in superscalar processors because of two reasons. First, data-flow architectures execute a data-flow graph where the association between the producer and the consumers of a value is explicit. To apply direct matching, we need to construct such a graph dynamically. Second, unlike the data-flow processors, most superscalar processors deal with general purpose code that has low to moderate parallelism. This type of code is very sensitive to any delays in the propagation of values. In order to be competitive, the graph has to be suitable for run-time generation by the hardware and it has to address the data fan-out problem efficiently.

2.1. The Fan-out problem

The fan-out problem is the representation and the implementation of the flow of values from their producers to their consumers. There are mainly three approaches to the problem. These are providing varying size destination lists (TTDA [1]), assuming a fixed fan-out per instruction and implementing the required fan-out by inserting *identity instructions* (ETS [6]) and finally assuming a fixed fan-out and blocking the instruction issue if issuing the new instruction will cause the fan-out limit to be exceeded (DTS [11]).

Using a varying length list is not suitable for superscalar processors since varying length lists are difficult to manage efficiently in the hardware. Similarly, the early algorithm DTS [11] is not suitable since it severely limits the parallelism that can be exploited. Finally, insertion of identity instructions is not desirable for superscalar processors since the need for a large data fan-out occurs when the instruction window becomes full. Insertion of identity instructions dynamically would fill-up the instruction window, resulting in reduced effective window size and consuming valuable functional unit bandwidth to execute the identity instructions.

Our solution to this problem uses our novel idea of *source operand to source operand forwarding* (SSF). Like ETS, we assume a fixed fan-out, but we give the capability to forward data to the source operands of the instructions as well. When an instruction executes, it sends its source operands to their next uses, as well as its result. Fig. 2 illustrates the flow of the values of `x` and `y` for various techniques.



(a) Varying length (b) Identity (c) SSF

Figure 2. Handling of data fan-out

As it can be seen, SSF handles the distribution of data values using a fixed fan-out per operand without introducing additional instructions. However, in its current form, it also restricts the amount of parallelism that can be exploited. This is because the propagation of a data value is delayed when one operand of an instruction arrives but the other operand of the instruction is missing. Since an instruction is scheduled for execution when both operands are available, the propagation of the available operand does not start until after the other operand is received. For example, in Fig. 2(c), instruction `I5` cannot start its execution until the long latency instruction `I2` completes and activates `I3`, although `I5`'s other operand would have been available long before the completion of I2. We have verified experimentally that this case occurs too often and it is detrimental to the performance. Fig. 3 compares the performance of SSF with a fan-out of two with the central window. As it can be seen, the loss of performance is quite large for larger issue widths. In order to eliminate the excessive delay, we need to allow data forwarding to continue when at least one of the operands is available. This approach leads us to the concept of treating the source operands and the instruction itself as independently schedulable entities. We therefore treat the source and result values uniformly and refer to each data value that needs to be propagated an *oplet.*

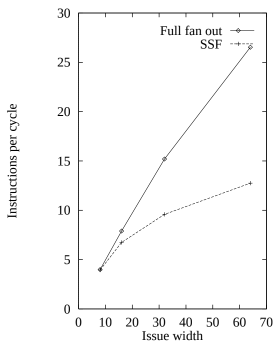


Figure 3. SSF-2 versus full fanout

The concept of oplets forms the basis of our graph. A dyadic instruction has three oplets, while a monadic instruction has only two. An oplet is an executable entity that has a tag indicating the status of the oplet, a value and a number of destinations that need this value. We define the execution of an oplet as the propagation of the value it carries to its destinations whereas execution of an instruction as consuming the input values, performing the operation indicated by the instruction and generating a result oplet. Therefore, the program execution is realized by the execution of the oplets of the graph and its instructions. Like any other dataflow style execution, both oplets and instructions can only execute when they have the required data values ready. Separation of the instruction operands into oplets has two significant consequences. It allows oplets to have their own *life time.* Therefore, an oplet receiving a data value can propagate it further without delays, given sufficient hardware resources. In other words, data flows freely through the forwarding edges. On the other hand, separate data operands requires additional links called *matching links* for joining them later. Revisiting the example in Fig. 2(c), we observe that the left operand of instruction `I3` can propagate the value to instruction `I5` although the instruction `I3` itself was still blocked waiting for the value of `y` from the divide instruction. As a result, the execution of `I5` can start as soon as `I4` is completed, while `I2` is still executing.

2.2. Dynamic Data Forwarding Graph

We now define the *Dynamic Data Forwarding Graph* (DDFG) formally.

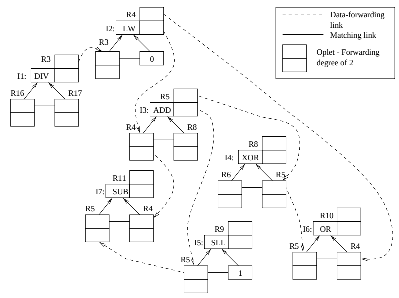
Definition: A DDFG is a directed graph $G = (V_{oplet} \cup V_{operation}, E_f \cup E_{match})$ where $V_{operation}$ is the set of nodes representing program instructions, V_{oplet} is the set of nodes representing instruction oplets, E_f is the set of data forwarding edges and E_{match} is the set of matching edges between the instructions and their oplets.

Definition: A DDFG has a *forwarding degree* F_d iff $\forall V \epsilon V_{oplet}, outdegree_f(V) \leq F_d$ where $outdegree_f$ is the number of forwarding edges emanating from V.

The DDFG deals with only those instructions which are in the instruction window and fully specifies the data driven execution in the instruction window. An example DDFG which has a forwarding degree of 2 is shown in Fig. 4. For the purposes of illustration, oplet nodes have been labeled with the register identifiers of the original program code.

```
DIV   R3,R16,R17
LW    R4,0(R3)
ADD   R5,R4,R8
XOR   R8,R5
SLL   R9,R5,1
OR    R10,R5,R4
SUB   R11,R5,R4
```

(a) Code



(b) DDFG

Figure 4. Sample code and its DDFG

There are many possibilities in the assignment of the forwarding edges in a DDFG, each leading to different instruction schedules. For the construction of the forwarding edges, we use a simple heuristic that works well. For each use of a value in the program order, we assign the use to a producer oplet breadth-first. This approach favors older instructions in the window, and rapidly distributes a value to its destinations since each level in the graph can distribute a value to increasingly more destinations.

2.3. DDFG Execution

The execution of DDFG is data-driven. An operation node can execute when it has both data values (assuming it is a dyadic operation), and an oplet can execute when it has one data value. Arrival of a data value at an oplet triggers the following sequence of events:

1. If the oplet has out-degree greater than zero, a copy of the data value is produced for each link and sent through each link.
2. A copy of the data value is sent through the matching link to the operation node and the oplet is deallocated.
3. Upon having both data values, an operation node performs the operation and sends the result value to the adjoining result oplet.
4. The result oplet sends the value through its forwarding edges and both the operation node and the result oplet are deallocated.

2.4. DDFG Construction

A DDFG is easily constructed as the program executes using an algorithm similar to renaming. For this purpose, an array of queues whose size is equal to the number of architectural registers is needed. Each queue entry holds a descriptor consisting of a counter and a pointer that points at the producer oplet which will have the value of a given register. Fetched instruction's source register identifiers are used to access the queue array to select the set of producer oplets for this register value. If the corresponding queue is empty, there are no pending values for this register and the value is provided to the instruction from the register file. Otherwise the descriptor at the head of the queue identifies the producer oplet that must be used for this consumer. A forwarding edge is set-up from the producer oplet to the current instruction oplet and the counter of the descriptor is incremented. If the value of the counter is greater than the degree of forwarding, the entry is removed from the queue. In any case, a new descriptor is formed which identifies the current instruction's oplet as a new producer and the descriptor is inserted in the queue. Once all the operands of an instruction are processed, a new descriptor is created for the result oplet, the queue corresponding to the result register is flushed and the new descriptor is inserted into the queue.

For a DDFG which has a degree of forwarding of one, the queue array becomes unneccessary and a table of size number of architectural registers is sufficient to generate the graph. When the size of each queue is unlimited, a balanced DDFG is obtained. While a balanced DDFG is ideal for the distribution of the values, in practice generated DDFGs will be unbalanced because of the limited queue sizes. We have observed that the queues must be as large as the degree of forwarding for achieving high performance but returns diminish rapidly beyond this size.

3. Data Forwarding Architecture

Although there are a number of ways to implement the semantics of the DDFG in the hardware, given that the benchmarks under consideration have low to moderate levels of parallelism, back-to-back execution of dependent instructions is quite significant. Our preliminary design of the architecture reflects this effort (Fig. 5). The fetch unit is responsible from fetching the instructions, decoding them and generating the DDFG. The functional units and their associated wait memories implement the portion of the instruction window where blocked instructions are kept. Although distributed to the functional units, wait memories share the same address space, and wait memory addresses are interleaved. The architecture manages the wait memory as a FIFO and allocates and frees the wait memory slots in program order similar to a reorder buffer. Key to the back-to-back execution of instructions is a small associative central structure called the *Match Unit* which keeps oplets which are likely to become ready in the current cycle. As a result, the match unit and the wait memories together form a hiearchy where oplets are brought into the match unit before they are needed from the slower wait memories.

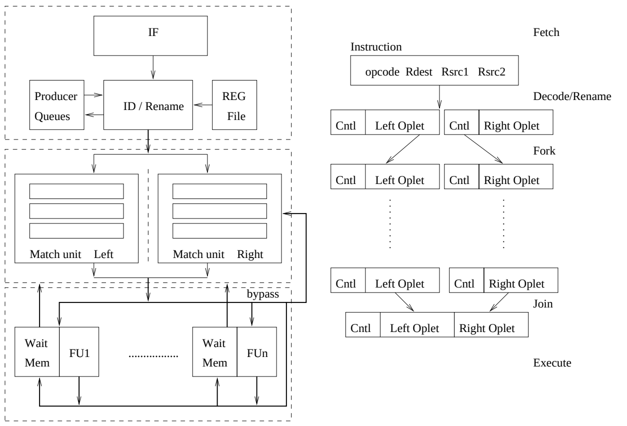


Figure 5. Superscalar Dynamic Data Forwarding Architecture (DDFA)

There are two sources of oplet flow into the match unit. The first is from the processor front-end through the fetch unit. Instructions sent from the fetch unit are split into two halves (i.e. *forked*), representing the left and the right oplet of the instruction and each half is placed into the corresponding side of the match unit. Similarly, one cycle before producing a result, functional units read the wait memory entry corresponding to their slot and send all the instruction oplets found to the match unit. While the instructions are executing, the match unit checks to see if both operands of an instruction are (or will be) available by comparing the oplets on the two sides of the match window. If this is the case, the two halves are *joined* and sent to a functional unit for execution. In this manner back-to-back execution of instructions is easily realized.

Oplets of instructions which are not sent to the functional units are stored back to the wait memories. When both operands of an instruction are missing, the control parts of the oplets are written into the wait memory entries of their producers. If only one operand is missing and the other is available, the control part (CNTL) of the available operand is marked to indicate that the operand is available and the oplet is written to the slot of the producer for the missing instruction. A copy of the oplet is sent to a functional unit to forward its source operand to other instructions. This way, the match unit always contains only the oplets of the instructions which may become ready and operates as just another pipeline stage in the architecture.

To completely eliminate structural hazards due to matching, the match unit should have sufficient entries to accomodate the newly fetched oplets as well as the oplets activated by completing instructions. The match unit size is determined by the issue width (I_w) and the degree of forwarding (F_d). The theoretical maximum size of the match unit in oplets is hence given by:

`Match unit size=` $I_w * (2 + F_d * 3)$

In practice, even the peak demand for match unit entries is well below the theoretical maximum. This is because, the average number of destinations across all instructions executed in a given program cannot exceed two, even when arbitrary fan-out is permitted [6].

4 Evaluation

We have evaluated the effectiveness of the dynamic data forwarding approach using detailed cycle level simulators. Simulators have been described for the MIPS ISA using the ADL [4] language and generated from ADL descriptions automatically.

Functional Unit	Latency(cycles)
Load	2
Integer division	8
Integer multiply	4
Other integer	1
Float multiply	4
Float addition	3
Float division	8
Other float	2

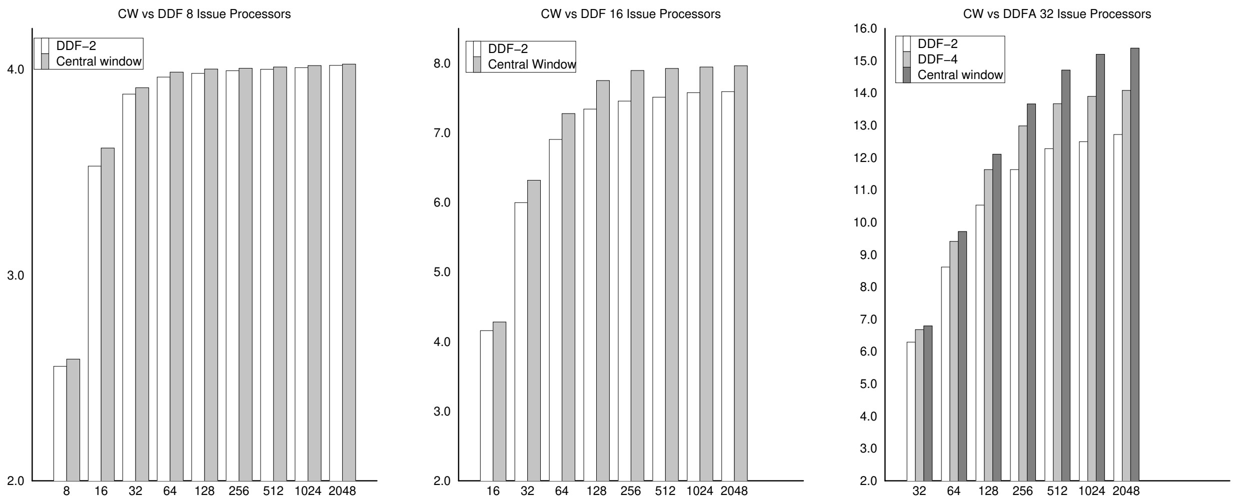
Table 1. Functional Unit Latencies

In the simulations, we assumed perfect branch prediction, perfect instruction fetch bandwidth and perfect memory disambiguation. However, we assumed realistic functional unit latencies (Table. 1) as assuming unit latencies would bias the study in favor of dynamic data forwarding. Given these assumptions, we varied the instruction window size and the issue width and compared the performance of various approaches using SPEC95 benchmarks. Benchmarks have been compiled using gcc version 2.7.0 with the optimization flags -O3. All programs were run to completion using the SPEC95 test inputs. However, in a few cases the input sets were modified to have a smaller data set. As a baseline processor, we used a superscalar processor based on Tomasulo's algorithm [9] which implements the full data fan-out using broadcasting and a central instruction window (CW).

The experiments consist of two sets. The first set studies the performance of the DDFA which assumes a match unit size set at the theoretical maximum. The purpose of this experiment is to decide the degree of forwarding necessary for a DDFG in order to have competitive performance to full data fan-out and to show that even at small degrees of forwarding there is little performance loss at a given instruction window size compared to the central window implementation which implements full data fan-out. In these experiments, the DDFA which has a forwarding degree of 2 is called DDF-2, and the DDFA which has a forwarding degree of 4 is called DDF-4. When we examine the results for 8 issue processors (Fig. 6(a)) we observe that a forwarding degree of 2 captures most of what can be extracted with a central window processor.

The results for 16 and 32 issue machines for the SPEC95 averages are shown in figures Fig 6(b) and Fig. 6(c). For issue widths upto 16 most of the performance of CW can be also be captured using a forwarding degree of 2. For example, at a window size of 256 the central window is only 6 percent better than DDF-2. Given the cost of increasing the degree of forwarding, the payoff is very little for forwarding degrees greater than 2 at these issue widths. However, at bigger issue rates such as 32, the effective window size rapidly increases. For 32 issue the effective window size is around 1024 entries although a window size of 2048 still provides some measurable performance improvement. For a window size of 1024, CW is faster than DDF-2 by about 21 percent, and faster than DDF-4 about 9 percent. More significantly, the required degree of forwarding increases slowly and a linear increase in the forwarding degree with respect to the issue width always matches the central window performance.

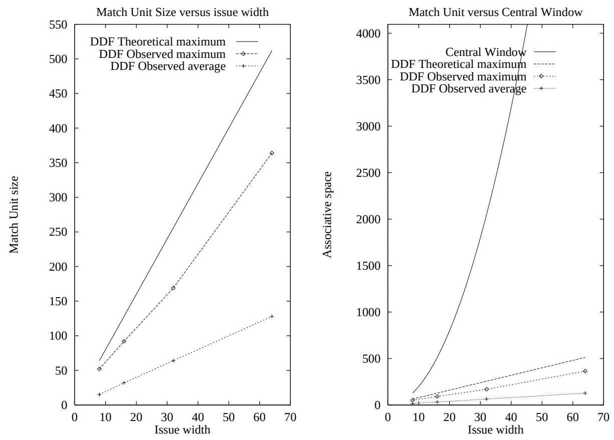
In the second set, we measured the actual match unit size utilized to demonstrate that in practice the match unit size is well below the theoretical maximum.



(a) 8 issue (b) 16 issue (c) 32 issue

Figure 6. IPC values for DDFA and CW

Fig. 7(a) shows the experimental results for the match unit size. Fig. 7(b) compares the required associative space with that of a conventional central window as a function of the issue width. As it can be clearly seen, DDFA approach is quite effective in reducing the amount of associative space that is needed to implement competitive performance.



(a) Match Unit Size (b) Associative Space

Figure 7. Match Unit

References

[1] Arvind, V. Kathail, and K. Pingali. A dataflow architecture with tagged tokens. Technical Report 174, MIT, 1980.

[2] E. Hao, P.-Y. Chang, M. Evers, and Y. N. Patt. Increasing the instruction fetch rate via block-structured instruction set architectures. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 191–200, December 1996.

[3] M. H. Lipasti and J. P. Shen. Superspeculative microarchitecture for beyond ad 2000. *IEEE Computer*, 30(9):59–66, 1997.

[4] S. Önder and R. Gupta. Automatic generation of microarchitecture simulators. In *IEEE International Conference on Computer Languages*, pages 80–89, Chicago, May 1998.

[5] S. Palacharla, N. P. Jouppi, and J. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th International Conference on Computer Architecture*, pages 206–218, June 1997.

[6] G. M. Papadopoulos. Implementation of a general purpose dataflow multiprocessor. Technical Report TR–432, MIT, 1988.

[7] Y. N. Patt, S. J. Patel, M. Evers, D. H. Friendly, and J. Stark. One billion transistors, one uniprocessor, one chip. *IEEE Computer*, 30(9):51–57, 1997.

[8] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 24–34, December 1996.

[9] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11:25–33, 1967.

[10] S. Vajapeyam and T. Mitra. Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. In *Proceedings of the 24th International Conference on Computer Architecture*, pages 1–12, June 1997.

[11] S. Weiss and J. E. Smith. Instruction issue logic in pipelined supercomputers. *IEEE Transactions on Computers*, C-33:1013–1022, November 1984.