Performance Evaluation of a Microarchitecture With Multiple Flows of Control

Francisco Santos[1], Rafael R. Santos[4], Anna Dolejsi Santos[3], Eliseu M. C. Filho[1], Philippe O. A. Navaux[2]

[1] COPPE-Universidade Federal do Rio de Janeiro
Rio de Janeiro, RJ, Brasil - {fcsantos, eliseu}@cos.ufrj.br
[2] Universidade Federal do Rio Grande do Sul
Porto Alegre, RS, Brasil - navaux@inf.ufrgs.br
[3] Universidade Federal Fluminense
Niteroi, RJ, Brasil - annads@dcc.uff.br
[4] Universidade de Santa Cruz do Sul
Santa Cruz do Sul, RS, Brasil - rrsantos@inf.ufrgs.br

Abstract—
This work presents a new approach to exploit the Instruction-Level Parallelism. In the MULFLUX microarchitecture each branch instruction spawns two different flows, corresponding to the two possible branch paths. Instructions from both flows are executed speculatively in parallel. Cycles consumed by discarded instructions are not visible, because the correct instructions also have been executed. The concept of multiple flows of control as exploited in this research work has provided promising results, with performance gains of up to 109% for a configuration supporting up to 16 active flows. Among the configurations considered here, the configuration with a maximum of 4 active flows exhibited the best compromise between performance and resource replication: it provided a performance gain of 105% and full dispatch width utilization of up to 41%.

Keywords—
Multiple Flows, Superscalar, Speculative Execution

I. Introduction

Modern high-performance microprocessors employ a pipelined superscalar architecture [1] to exploit Instruction-Level Parallelism (ILP). The performance of a superscalar architecture is significantly constrained by control dependences [2]. Control dependences restrict performance by reducing the utilization of the fetch bandwidth and by avoiding instructions from different basic blocks to enter the dynamic execution window.

Lam and Wilson [3] evaluate two alternatives for relaxing control flow constraints. These are *control dependence analysis* and *multiple flows of control.* Control dependence analysis is a software technique that discovers parallelism from different regions of code, each with its own control flow. The concept of following multiple flows of control is viewed as the architectural support to fully exploit the additional parallelism exposed by control dependence analysis.

In this work, flows of control are not explicitly associated with statically defined pieces of code. Instead, they are implicitly created by branch instructions. More precisely, each branch instruction spawns two different flows, corresponding to the two possible branch paths. Instructions from both flows are executed speculatively in parallel. Once a branch outcome is determined, instructions in the wrong path are discarded, as in the conventional speculative model. However, cycles consumed by discarded instructions are not visible, because the correct instructions have been executed in parallel. In addition, the correct flow of control can advance considerably while previous branches are blocked due to (data) dependences. In the conventional speculative model, the single flow of control can also advance in the presence of blocked branch instructions but, if the first dispatched branch is mispredicted, this advantage is lost.

Other works have also pursued execution models based on multiple flows of execution. Wallace *et al.* [4] devised a scheme called *threaded multiple path execution*, which actually employs a simultaneous multithreading [5] architecture to speculatively execute multiple branch paths. Whenever the number of executing threads is less than the total number of hardware contexts, the spare contexts are used to fetch and execute along the less likely path of a branch instruction. Klauser *et al.* [6] proposed an execution model called *selective eager execution* and its corresponding substrate, the *PolyPath processor.* As in our work, their purpose is to overcome mispredicted branch penalties by executing both paths originated by a branch instruction. Chronologically, the present work has been one of the first to investigate a practical execution model based on multiple flows of control [7].

The remainder of this paper is organized as follows. Section 2 introduces the concept of multiple flows of control. Section 3 describes MULFLUX, an architecture supporting the parallel execution of multiple flows of control. Section 4 shows experimental results assessing the performance of this architecture. Section 5 concludes the paper listing potential areas for future work.

II. A Model of Multiple Flows of Control

The parallel execution of multiple flows of control can be represented through a binary tree, called *Dynamic Instruc-*

tion Flow Tree (DIFT). Each node in the DIFT corresponds to a fetched branch instruction which has not been resolved yet. The arcs of the DIFT represent the dynamic execution flow along the paths that originate from the branch instructions. The DIFT is called *dynamic* because its topology changes during the program execution, according to the resolution of branch instructions. Figure 1 shows a simple DIFT. Two instruction flows, s_0 and s_1 originate from the branch instruction b_0. By convention, the dynamic execution flow leading to a branch instruction continues along the not-taken path of the branch. This flow is referred to as the *parent flow*. The new instruction flow originated from a branch instruction is called the *child flow*, and it comprises the instructions along the taken path of the branch.

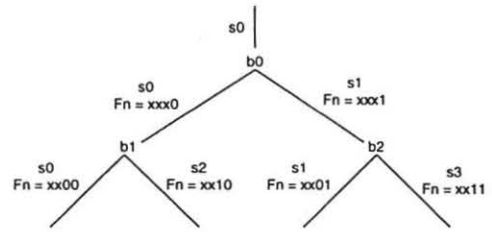


Fig. 1. Example of a Dynamic Instruction Flow Tree (DIFT).

In our model, branch instructions are executed sequentially, according to the order in which they are fetched. When the outcome of a branch instruction is determined, the flow along the wrong branch path should be cancelled. In addition, flows descending from the wrong flow should also be cancelled. Flow cancellation requires some sort of identification, from which it is possible to determine the flows that should be cancelled.

For that purpose, a *Flow Number* (F_n) is assigned to each flow in the DIFT. The flow number has a bit for each level in the DIFT, with the least significant bit corresponding to level $l = 1$ (DIFT levels are counted starting from the root). The number of bits in the flow number is given by the maximum allowed DIFT depth (which, by its turn, depends on the available architectural resources). A bit 0 indicates an instruction flow along the not-taken path of a branch, while a bit 1 indicates a flow along the taken path. As Figure 1 shows, from branch b_0, flow s_0 has a flow number F_n = xxx0, as it runs along a not-taken path in level $l = 1$. The child flow s_1 has F_n = xxx1, because it runs along a taken path in level $l = 1$. In the above, x means a don't care.

Flow numbers are used in the following way during flow cancellation. If the outcome of a branch instruction is not-taken, all the descending flows with least significant bit 1 in their flow numbers are cancelled. Similarly, if the branch outcome is taken, descending flows with least significant bit 0 in the flow numbers are cancelled. After cancellation, the remaining flows should be renamed. Flow renaming is achieved by simply shifting flow number bits one position to the right. A new DIFT is obtained after flow cancellation and subsequent flow renaming. In the example shown in Figure 2, branch b_0 is taken. Dashed lines in Figure 2(a) indicate the cancelled flows. Figure 2(b) shows the resulting DIFT after flow renaming. Notice that, as branches are executed sequentially in order, the branch currently being processed is the one in the DIFT root.

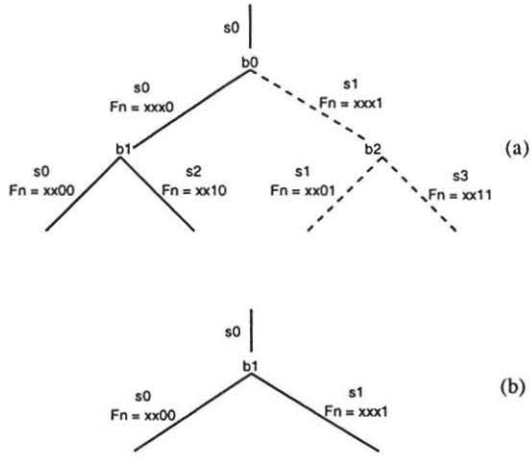


Fig. 2. Flow cancellation and flow renaming.

A. *Inter-flow Data Dependences*

Data dependences may occur among instructions belonging to the same flow. These data dependences will be referred to as *intra-flow data dependences*. In addition, the simultaneous execution of multiple flows introduces data dependences involving instructions from different flows. Data dependences in this new class will be called *inter-flow data dependences*. Figure 3 shows an example of inter-flow data dependence.

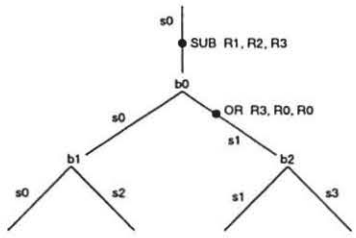


Fig. 3. Example of inter-flow data dependence.

The SUB instruction in the parent flow s_0 writes into register R3, while the OR instruction in child flow s_1 reads from that register. If no preceding instruction in flow s_1 writes into R3, the value read from R3 by the OR instruction should be the one produced by the SUB instruction. The OR instruction in s_1 can be executed only after the SUB instruction is

completed, therefore establishing a data dependence between flows s_0 and s_1.

Each flow has a separate state. Therefore, data produced in one flow usually does not interfere in the state of other flows. However, it may happen that data produced by instructions in a certain flow need to be visible to instructions in other flows. Figure 4(a) shows an example. In this figure, the SUB instruction in flow s_0 writes into register R3. The value written into R3 should be visible to instructions in flows s_0 and s_2. In contrast, the value produced by SUB should not be visible to those instructions in flows s_1 and s_3. Actually, data visibility among flows is a consequence of, and is determined by, inter-flow data dependences.

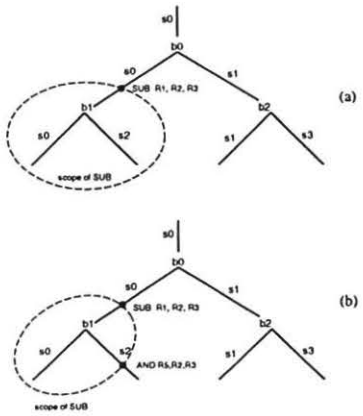


Fig. 4. Data visibility among multiple flows.

Therefore, a rule is necessary to control data visibility among instructions from the multiple flows. The following scope rule is defined: the scope of instructions in flow s comprises the instructions in the flows descending from s. Or, in other words, data produced in a certain flow should be visible only to the descending flows.

A priori, the scope of a certain instruction covers *all* instructions in descendant flows. However, the scope can "shrink" as execution evolves. In the example shown in Figure 4(b), the AND instruction in flow s_2 also writes into register R3. Now, subsequent instructions in s_2 reading from R3 should get the value produced by the AND instruction, instead of the value from the SUB instruction in flow s_0 (in such situations, intra-flow dependences override inter-flow dependences). Thus, once the AND instruction is fetched, the scope of the SUB instruction becomes restricted to those instructions preceding the AND instruction. For this reason, the scope of instructions is more precisely referred to as *dynamic* scope.

III. The MULFLUX Architecture

The previous section described the abstract model of multiple flows of execution, as considered in this work. This section presents an architectural substrate to support that model. This substrate, called the MULFLUX *architecture*, requires the same mechanisms found in conventional, speculative superscalar architectures. But, in order to support multiple flows of execution, such mechanisms were extended in the MULFLUX architecture in the following ways: (1) the fetch mechanism is extended to access instructions from multiple flows, and to create, cancel and rename flows. Notice that branch prediction is completely eliminated; (2) the decode and dispatch mechanisms are extended to handle inter-flow data dependences and to implement the scope rule; and (3) the commit mechanism is extended to maintain the states associated with multiple flows of control.

A. Executing Multiple Flows of Control

The MULFLUX architecture is built around an extension of the register renaming and speculative execution mechanism found in the MIPS R10000 processor [8]. Before discussing the extension to multiple flows of control, it is adequate to briefly describe the operation of the basic register renaming mechanism. It comprises three components: (1) the *mapping table* keeps the correspondence between the id of a logical register, indicated in the instruction, and the id of the physical register which actually stores data; (2) the *active list* contains id's of physical registers that are no longer mapped to logical registers and which can not be reused because the corresponding instructions have not been completed yet; and (3) a *free list*, which contains id's of physical registers that are currently unmapped.

Let us consider that instruction `ADD R1,R2,R3` arrives at the renaming stage and that, at this moment, the mapping table has the mappings $1 \rightarrow 32$ (i.e., logical register 1 is mapped to physical register 32), $2 \rightarrow 33$, $3 \rightarrow 34$. In addition, suppose that logical id 40 is at the head of the free list. Logical source registers are renamed to physical registers according to the current mapping table state, while the logical destination register is renamed to a physical register indicated by the id retrieved from the free list. Therefore, after renaming, the instruction becomes `ADD R32,R33,R40`. As part of the renaming process, the mapping table is updated with the new mapping $3 \rightarrow 40$, while the previous mapping $3 \rightarrow 34$ is saved in the active list. Insertions in the active list follow the static instruction order.

The id of a physical register returns from the active list to the free list only when two conditions are met: (1) the id is at the head of the active list and (2) the instruction which moved the id from the mapping table to the active list has been executed. These two conditions ensure that all instructions that read a physical register have done so before its id returns to the free list and the register is reutilized.

To support speculative execution, a scheme is necessary to discard the mappings which are created by instructions following a mispredicted branch. In other words, it is necessary

to recover the mapping table to the state existing before the access of the mispredicted branch. This can be achieved by using the mappings saved in the active list. When a mispredicted branch is found, mappings subsequent to the branch entry in the active list are copied back to the mapping table. When a mapping is undone, the physical register id is moved from the mapping table to the free list. Notice that speculative results are implicitly discarded when mappings subsequent to a mispredicted branch are undone.

This scheme does not require data storage replication to keep the correct architectural state: a single register file keeps both the speculative and the correct state. For such reason, the mechanism smoothly extends to support multiple flows of execution, as described next.

B. Extension to Multiple Flows of Control

As Figure 5 shows, the extended register renaming mechanism consists of replicated mapping tables, a single active list and a single free list. The number of mapping tables is one of the factors that determines the maximum number of active flows.

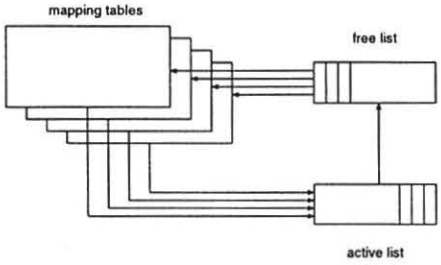


Fig. 5. The extended register renaming mechanism.

A mapping table is allocated when a child flow is created. It keeps the mappings established by instructions belonging to the child flow. When a logical destination register is mapped to a physical register, a pair (F_n, R_d) is inserted into the active list. Here, F_n is the instruction's flow number and R_d is the physical register id displaced from the mapping table by the new mapping.

The mapping table allocated to the child flow inherits the mappings stored in the parent's mapping table. Inter-flow dependences are thus automatically satisfied, because instructions in the child flow will read the same physical registers in use by the parent flow at the moment of child flow creation. The scope rule is satisfied due to the use of multiple mapping tables and the single free list. If instructions from different flows write into the same logical register, this register will be mapped differently for the two instructions, as the mapping tables are distinct and the physical register id is retrieved from a single free list.

Flow cancellation involves discarding the dispatched instructions belonging to that flow. This is done using the flow numbers stored in the active list. An instruction is labeled as discarded if the least significant bit of its F_n in the active list matches the least significant bit of F_n of the flow being cancelled.

C. Handling id Duplicates

In a register renaming mechanism, it is necessary to keep the uniqueness of physical register ids. However, physical ids are replicated when a mapping table inherits information from another table. These duplicates can reach the free list, therefore breaking physical id uniqueness, in two situations: (1) duplicates are inserted into the active list if instructions from sibling flows write into the same logical register, before one of the flows is cancelled. From the active list, duplicates may return to the free list; and (2) a duplicate physical id may return to the free list from a mapping table associated with a cancelled flow.

These two special cases must be handled in order to guarantee physical id uniqueness. Both can be treated by using a counter associated with each physical register, which indicates the number of duplicate id's for that physical register. Whenever a newly allocated mapping table is initialized, the counters corresponding to the copied physical id's are incremented. When a physical id is removed from the active list, the corresponding counter is decremented. In this case, the id returns to the free list if the counter is zero after being decremented. When a flow is cancelled, the counters corresponding to the physical id's found in the flow's mapping table are all decremented. Again, a physical id returns to the free list only if the counter becomes zero after decrement.

D. Intra-flow Data Dependences

Besides handling inter-flow data dependences, the register renaming mechanism eliminates false intra-flow data dependences. An additional mechanism is necessary to handle true intra-flow data dependences.

Instructions are dispatched to *issue buffers* attached to the functional units. Instructions are issued out of order to the functional units according to operand availability. The issue buffer has an *invalid bit* for each instruction's operand. An invalid bit is also associated with each physical register. This bit is set when the physical register is mapped to a logical destination register, thus indicating that a new value will be written in that physical register.

During dispatch, the invalid bit in the issue buffer is set if the invalid bit of the corresponding physical source register is also set. Upon instruction completion, the invalid bit of the physical destination register is reset. In addition, the physical destination register id is broadcast to the issue buffers. The invalid bits of matching source register id's in the issue buffers are then reset, possibly enabling instructions for issue. Input operands are read from physical source registers when an instruction is issued to a functional unit.

E. Fetching from Multiple Flows of Control

Instructions from a certain flow are stored into a *flow fetch buffer* (FFB), shown in Figure 6. Each FFB comprises: a *busy bit*, which indicates whether the FFB is allocated to a flow; a *flow number register*, which contains the flow number; a *program counter register*; and an *instruction FIFO*.

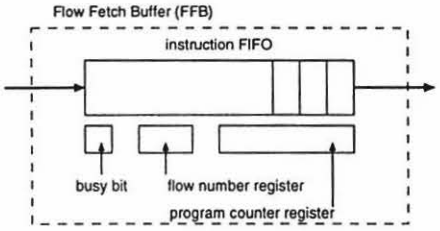


Fig. 6. Instruction fetching in the MULFLUX architecture.

A child flow is only created if there is a free FFB for it. Otherwise, the branch instruction is assumed as not-taken and fetching continues along the parent flow. As part of child flow creation, the program counter register of the allocated FFB is initialized with the branch's target address, and the flow number register is initialized with the child's flow number. Finally, the busy bit is set. The parent flow continues with the FFB already allocated to it.

FFB deallocation uses the information in the flow number registers. Recall that flow cancellation is decided when a branch instruction is processed; either the parent flow or the child flow, and the appropriate descendants, have to be discarded. If the parent flow has to be discarded, all FFBs with a 0 as the least significant bit in the flow number register are released (i.e., their busy bits are reset). If the child flow has to be discarded, the FFBs deallocated are those with a 1 as the least significant bit in the flow number register.

F. Organization of the Mulflux Architecture

The mechanisms described so far appear as components of a pipeline with six stages: fetch (IF), decode (DC), dispatch (DS), issue (IS), execute (EX) and write-back (WB).

Stage IF comprises the multiple flows fetch mechanism described in the previous subsection. In addition to decoding, stage DC performs register renaming, mapping table allocation and mapping table initialization. Renamed instructions are inserted into *dispatch queues*, located in stage DS. This stage sends instructions from the dispatch queues to the appropriate *issue buffers*, stalling when there is no free issue buffer.

Control transfer instructions are executed in stage DS by a *branch unit*. Unconditional control transfer instructions are processed by the branch unit in the same cycle they are dispatched. Conditional transfer instructions are also processed in the dispatch cycle if the required operands are available. Otherwise, the branch instruction waits in one of the issue buffers attached to the branch unit. The branch unit executes branch instructions sequentially and in the order they enter the issue buffers. As instructions from the same flow are dispatched in order, branch instructions along a flow (and, in particular, along the correct flow) are executed in order.

Stage IS updates the status of instructions in the issue buffers according to instruction completion. If the appropriate functional unit is free, stage IS selects a ready instruction from the associated issue buffers, reads source data from the register file and sends instruction and data to the functional unit.

Stage EX has multiple integer functional units, a data memory access unit and a floating-point unit. Upon receiving the result from a functional unit, stage WB sends the physical destination register id back to stage IS and writes the result into the appropriate physical register. It also sets the done bit in the active list entry corresponding to the completed instruction.

Further implementation details of this architecture can be found in [9]. In particular, we show how to implement duplicate id control through a mechanism functionally equivalent to the counter-based scheme explained before, but which uses simpler bit vectors instead of actual counters.

IV. Evaluation of Mulflux

A. Experimental Framework

In order to evaluate the performance of the execution model proposed here, we have built a trace-driven simulator (hereafter called MULFLUX *simulator*) for the architecture described in the previous section. In our experiments, traces were generated by using an execution-driven simulator of a scalar, pipelined implementation of the SPARC architecture [10].

The performance gains obtained with the MULFLUX architecture were evaluated by comparing its (average) ipc with the ipc delivered by a conventional superscalar architecture. The latter is hereafter referred to as the *reference architecture*. Our reference architecture resembles the PowerPC 604 and PowerPC 620 architectures [11], [12]. It has the same number of pipeline stages found in MULFLUX architecture. It employs the Tomasulo algorithm to handle data dependences, a BHT [2] for dynamic branch prediction and a reorder buffer with future register file [2] to support speculative execution. The dispatch width is four instructions per cycle. Finally, the reference architecture has four integer functional units and one pipelined memory functional unit, with eight reservation stations for each unit. It also includes a 32 KB instruction cache and a 32 KB data cache. The ipc of the reference architecture was obtained by using another trace-driven simulator, which also uses the trace files generated by the SPARC simulator as input.

In the experiments, we have used eight integer programs

from SPEC92 and SPEC95. These programs were compiled into SPARC executable code by using the GNU gcc-2.7.2 C compiler, with -O optimization flag.

B. Results and Analysis

Performance of MULFLUX was measured as a function of two architectural parameters: the maximum number of active flows and the dispatch width. The values for the maximum number of active flows are: 2, 4, 8 and 16 flows. The values for the dispatch width are 2 times and 4 times the maximum number of active flows. For instance, for a maximum of 2 active flows, we consider dispatch widths of 4 and 8 instructions/cycle.

The remaining parameters of the MULFLUX architecture were fixed as follows: per-flow fetch width is 4 instructions/cycle; FFBs and dispatch queues have 16 entries each; the speculation depth is 16 branches; there are 4 integer units and 1 memory unit, each one with 8 issue buffers; and there are 5 result buses, shared by the functional units.

Figure 7 depicts the performance of the MULFLUX architecture. The white bar corresponds to the performance of the reference, single-flow superscalar architecture. The other two bars indicate the MULFLUX performance, for dispatch widths (DW) of 2 and 4 times the maximum number of active flows.

When there are at most 2 active flows, the smallest gain was 2.5% (`m88ksim` program with DW = 4), while the highest gain was 81.6% (`vortex` program with DW = 8). By changing the dispatch width from 4 instructions/cycle to 8 instructions/cycle, the performance gain ranges from 1.6% (`m88ksim`) to 12.3% (`compress`).

When we allow at most 4 active flows, performance gain ranges from 27.2% (`eqntott` with DW = 8) and 105.3% (`vortex` with DW = 16). By increasing the dispatch width from 8 instructions/cycle to 16 instructions/cycle, the maximum observed gain is 2.3%.

When the maximum number of active flows is either 8 or 16 flows, we observe performance returns smaller than in the previous two cases. With 8 active flows and DW = 16, the minimum performance gain is 27.2%, the same measured with 4 active flows. The maximum gain is 109.8%, close to the highest gain obtained with 4 active flows (105.3%). With 8 active flows, an increase in the dispatch width resulted in a performance gain of at most 1.55%.

An important effect is observed when the maximum number of active flows is 16 flows. Performance decreases relative to the case with 8 active flows, for the following programs: `espresso`, `eqntott`, `gcc`, `m88ksim` and `ijpeg`. The decrease is only 0.52% for the `espresso` program, but it is as high as 11.8% for the `gcc` program.

This behavior comes from the fact that the number of resources, mainly issue buffers, is not adequate for the large number of active flows. In consequence, instruction dispatch frequently stalls due to unavailable issue buffers. In addition, destructive interference among flows increases when there is a large number of active flows. Instructions from wrong flows interfere by being dispatched before instructions belonging to correct flows. Wrong instructions occupy issue buffers, delaying the dispatch of correct instructions. In this case, cycles consumed with the execution of wrong instructions become visible. The probability of such interference increases with the number of active flows.

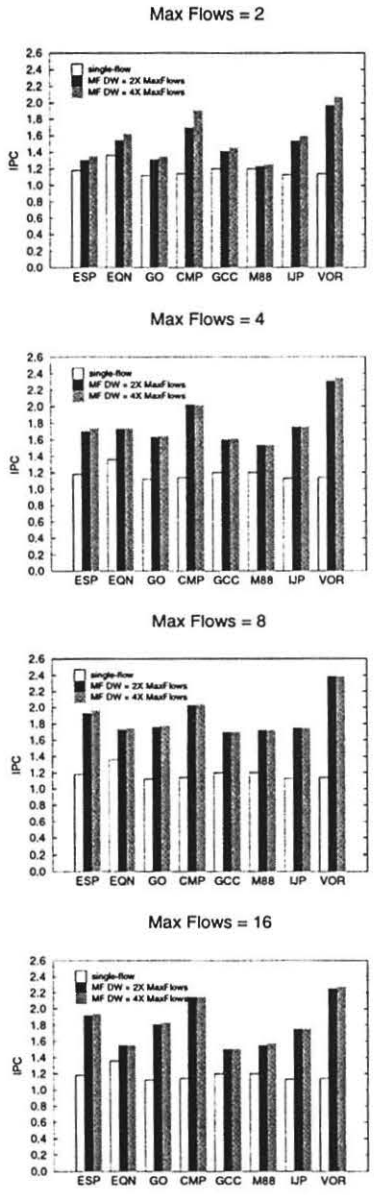
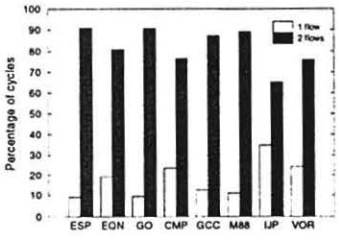


Fig. 7. Performance gains of the MULFLUX architecture.

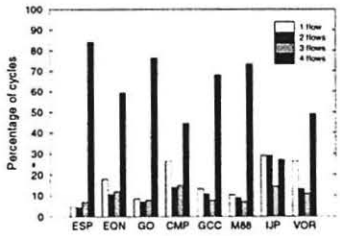
We notice that performance is more sensitive to the dispatch width as the number of active flows is smaller. As we

have seen, with at most 2 active flows, performance increases by up to 12.3% when the dispatch width is doubled. However, with 8 active flows, the increase obtained by doubling the dispatch width is only 1.55%.

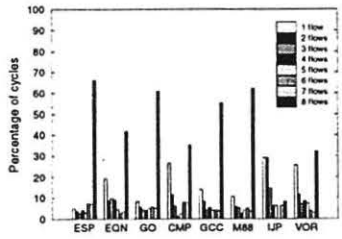
One important measure is the average number of active flows active at a certain moment. This information is useful to optimize resource replication in the architecture. Figure 8 shows the distribution of active flows, when the maximum number of flows is 2, 4 and 8 flows. Each bar indicate the percentage of the total number of cycles in which we have a certain number of active flows. The plots show that the maximum flow capacity is utilized most of the time. In the three cases and for almost all programs, the maximum number of active flows is reached over 60% of the time. This behavior is attributable to the high dynamic count of branch instructions in integer programs. The only observed exception is the `ijpeg` program, when the maximum number of active flows is either 4 or 8 flows. For that program, only 3 flows are active most of the time.



Max Number of Flows = 2.



Max Number of Flows = 4.



Max Number of Flows = 8.

Fig. 8. Distribution of the number of active flows.

Another important measure is the dispatch width utilization. This information is useful for the appropriate dimensioning of the dispatch width, a parameter critical to the implementation cost. Tables I, II and III show the the utilization of the available dispatch width when the maximum number of active flows is 2, 4 and 8 flows, respectively. The dispatch width is always 2 times the maximum number of flows (i.e., the tables correspond to dispatch widths of 4, 8 and 16 instructions/cycle respectively). Each column in a table indicates the percentage of the total number of cycles in which a certain number of instructions were dispatched.

TABLE I
DISPATCH WIDTH UTILIZATION FOR MAXFLOW = 2.

Program	Percentage of Dispatched Instructions			
	1	2	3	4
espresso	1.6%	3.6%	4.9%	42.8%
eqntott	2.5%	3.6%	3.9%	57.3%
go	3.3%	3.6%	4.7%	48.7%
compress	16.4%	10.5%	2.3%	50.3%
gcc	5.5%	5.0%	5.4%	50.0%
m88ksim	3.7%	3.4%	6.8%	43.1%
ijpeg	0.0%	3.7%	8.2%	56.0%
vortex	2.9%	2.7%	3.1%	66.3%

TABLE II
DISPATCH WIDTH UTILIZATION FOR MAXFLOW = 4.

Program	Percentage of Dispatched Instructions							
	1	2	3	4	5	6	7	8
espresso	9.9%	9.7%	7.6%	5.6%	2.6%	2.7%	2.8%	35.8%
eqntott	11.3%	5.2%	2.9%	11.2%	3.8%	2.8%	3.1%	28.0%
go	5.0%	3.8%	2.7%	8.2%	4.8%	2.7%	3.1%	41.4%
compress	10.8%	9.9%	8.3%	12.5%	7.3%	7.6%	5.3%	19.1%
gcc	6.2%	4.4%	3.9%	12.6%	5.6%	4.4%	3.7%	34.8%
m88ksim	8.7%	1.9%	3.0%	8.7%	2.7%	5.6%	3.4%	37.6%
ijpeg	6.0%	2.1%	2.1%	25.0%	0.0%	0.3%	2.4%	31.0%
vortex	2.8%	2.6%	2.6%	19.4%	3.7%	2.7%	3.2%	41.1%

With a maximum of 2 flows (and a dispatch width of 4 instructions/cycle), the full dispatch width is utilized from 43% to 66% of the cycles. For a maximum of 4 active flows (dispatch width of 8 instructions/cycle), we still observe considerable utilization of the full dispatch width. In this case, full utilization occurs from 19% to 41% of the cycles. However, with a maximum of 8 flows (dispatch width of 16 instructions/cycle), full width utilization is smaller, occurring only on 3% of the cycles for the `compress` program, and at most during 37% of the cycles for the `espresso` program (see Table III on the next page).

V. CONCLUDING REMARKS

The idea of simultaneously pursuing multiple branch paths has been considered of difficult implementation, as it requires hardware resources exponential to the number of active flows. However, during the last decade, advances in integration technology have made possible the implementation of complex microarchitectures. Several parallel microarchitectures that require aggressive resource replication have been proposed recently. Compared with those, the MULFLUX

TABLE III
DISPATCH WIDTH UTILIZATION FOR MAXFLOW = 8.

Program	Percentage of Dispatched Instructions															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
espresso	4.8%	4.3%	3.6%	6.4%	2.8%	2.6%	2.7%	2.0%	1.7%	1.9%	1.9%	1.8%	1.8%	1.9%	1.5%	37.3%
eqntott	8.7%	7.2%	4.6%	10.9%	3.6%	2.1%	2.8%	4.0%	3.5%	1.0%	1.2%	2.0%	1.1%	0.8%	0.5%	11.3%
go	7.4%	5.2%	4.2%	6.9%	3.0%	2.6%	2.2%	3.1%	3.4%	2.2%	2.4%	3.6%	2.8%	1.8%	2.0%	21.6%
compress	12.0%	6.0%	5.2%	8.8%	7.5%	6.2%	3.9%	7.2%	3.1%	3.0%	3.7%	1.5%	4.0%	1.8%	2.2%	3.1%
gcc	7.4%	5.6%	5.0%	11.4%	4.1%	2.7%	2.4%	3.9%	4.6%	3.2%	2.1%	2.2%	2.7%	2.3%	2.1%	16.7%
m88ksim	6.3%	4.2%	3.8%	10.0%	4.1%	3.8%	2.3%	2.8%	2.8%	2.9%	2.6%	3.7%	2.6%	2.4%	2.2%	22.8%
ijpeg	12.2%	2.1%	4.2%	16.7%	0.0%	3.9%	0.0%	4.4%	8.3%	4.4%	1.8%	0.0%	2.1%	0.0%	0.0%	6.2%
vortex	7.0%	3.8%	2.8%	18.3%	3.1%	2.1%	1.8%	4.4%	5.6%	2.6%	1.7%	2.4%	2.7%	2.0%	1.4%	15.4%

microarchitecture has a simpler implementation.

The concept of multiple flows of control as exploited in this research work has provided promising results, with performance gains of up to 109% for a configuration supporting a maximum of 16 active flows. Among those configurations here considered, configuration with a maximum of 4 active flows exhibited the best compromise between performance and resource replication: it provided a performance gain of 105% and full dispatch width utilization of up to 41%.

We intend to continue the research along several investigative tracks. In order to obtain better dispatch width utilization, it is necessary to determine the adequate balance between the number of active flows and machine parallelism. Another important aspect is the organization of the cache subsystem to support instruction accesses from different flows. Finally, in order to reduce the average number of active flows, we intend to use confidence assignment to control flow creation.

REFERENCES

[1] Smith, J. E., Sohi, G. S., *The Microarchitecture of Superscalar Processors*, Proc. IEEE, Vol. 83, No. 12, Dec. 1995, pp. 1609–1624.

[2] Patterson, D., Hennessy, J., *Computer Architecture: A Quantitative Approach*, 2nd. Ed., Morgan-Kaufmann, San Francisco, CA, 1997.

[3] Lam, M. S., R. P. Wilson, *Limits of Control Flow on Parallelism*, Proc. 19th International Symposium on Computer Architecture, 1992, pp. 46–57.

[4] Wallace, S., Brad Calder, *Threaded Multiple Path Execution*, Proc. of the 25th International Symposium on Computer Architecture, 1998, pp. 238–249.

[5] Tullsen, D. M., S. J. Eggers, *Simultaneous Multithreading: Maximizing On-Chip Parallelism*, Proc. of the 22nd International Symposium on Computer Architecture, 1995, pp. 392–403.

[6] Klauser, A., A. Paithankar, *Selective Eager Execution on the PolyPath Architecture*, Proc. of the 25nd International Symposium on Computer Architecture, 1998, pp. 250–259.

[7] *Mulflux: Superscalar Architectures with Multiple Instruction Flows*, Research Project Proposal submitted to the Brazilian National Research Council, 1995.

[8] MIPS Technologies Inc., *MIPS R10000 Microprocessor User's Manual*, Mountain View, CA., 1995.

[9] *MULFLUX: A Superscalar Architecture with Multiple Flows of Control*, Proceedings of the ProTem-CC III Project Evaluation Workshop, May 1999. (also available at http://www.cos.ufrj.br/mulflux)

[10] Sun Microsystems, *The SPARC Architecture Manual*, Mountain View, CA, 1987.

[11] IBM Corp., *PowerPC 604 RISC Microprocessor Technical Summary*, IBM Order Number MPR604TSU-01.

[12] Diep, T. A., J. P. Shen, *Performance Evaluation of the PowerPC 620 Microarchitecture*, Proc. of the 22nd International Symposium on Computer Architecture, 1995, pp. 163–175.