URCS Technical Report 743
Dynamically Allocating Processor Resources between Nearby and Distant ILP*

Rajeev Balasubramonian[†], Sandhya Dwarkadas[†], and David H. Albonesi[‡]
[†] Department of Computer Science
[‡] Department of Electrical and Computer Engineering
University of Rochester
April, 2001

*Extended version of the paper in the 2001 International Symposium on Computer Architecture. This work was supported in part by NSF grants CDA–9401142, EIA–9972881, CCR–9702466, CCR–9701915, CCR–9811929, CCR–9988361, and CCR–9705594; by DARPA/ITO under AFRL contract F29601-00-K-0182; and by an external research grant from DEC/Compaq.

Abstract

Modern superscalar processors use wide instruction issue widths and out-of-order execution in order to increase instruction-level parallelism (ILP). Since instructions must be committed in order so as to guarantee precise exceptions, increasing ILP implies increasing the sizes of structures such as the register file, issue queue, and reorder buffer. Simultaneously, cycle time constraints limit the size of these structures, resulting in conflicting design requirements.

In this paper, we present a novel microarchitecture designed to overcome the limitations of a register file size dictated by cycle time constraints. Available registers are dynamically allocated between the primary *program thread and a* future *thread. The* future *thread issues and executes instructions when the* primary *thread is limited by resource availability. The* future *thread is not constrained by in-order commit requirements. It is therefore able to examine a much larger instruction window and jump far ahead to execute ready instructions. Results are communicated back to the* primary *thread by warming up the register file, instruction cache, data cache, and instruction reuse buffer, and by resolving branch mispredicts early. The proposed microarchitecture is able to get an overall speedup of 1.17 over the base processor for our benchmark set, with speedups of up to 1.64.*

1 Introduction

Dynamic superscalar processors perform register renaming and out of order issue in hardware to extract greater instruction-level parallelism (ILP) from existing programs. A significant performance limitation in such processors is the lack of forward progress in the midst of long latency operations (*e.g.*, cache misses). When this happens, it would ideally be most beneficial to execute other independent performance degrading instructions (long-latency loads, branch mispredicts). However, to find such independent instructions, the processor would have to examine a sufficiently large instruction window.

This problem cannot be solved by simply increasing the number of in-flight instructions, as it would require larger register files and reorder buffers that may impact critical timing paths. The register file, in particular, can often determine the cycle time and several approaches that attempt to balance latency and IPC have been proposed. The Alpha 21264 implements a clustered register file [14] in an attempt to reduce average latency. Similarly, register file caches have also been proposed [7] in order to access a smaller subset of registers in a single cycle. Both of these techniques, however, cause IPC degradation when compared to a single monolithic register file of the same size. A multi-cycle register file has its own problems - design complexity in pipelining a RAM structure, having two levels of bypass (which is one of the critical factors in determining cycle time [7, 21]), and reduced IPC because of longer branch mispredict penalties and increased register lifetimes. These problems are only exacerbated in an SMT processor, where the register file resources have to be shared by multiple threads. Further, as we move to smaller process technologies, the dominating effect of long wire delays will make it even more prohibitive to implement large register files in wide-issue machines [12, 21].

The fundamental reason why the register file size has such a large impact on the size of the instruction window, and hence performance, is that instructions can be renamed and dispatched only when there are free registers available. Registers are freed only when instructions commit, and instructions are committed in order. A single instruction that takes a long time to complete could stall the commit stage, thereby holding up all the registers and not allowing subsequent instructions to dispatch. During this period, the out-of-order execution core can only look at a restricted window of instructions to extract ILP. As the processor-memory gap increases, there will be an increasing number of long-latency loads, causing dispatch to frequently stall as it runs out of physical registers. Thus, there is a need for new approaches that allow for forward progress to be made without increasing the complexity of critical hardware structures.

In this paper, we present a novel architecture that uses the limited number of physical registers to dynamically trade nearby with distant ILP, while still maintaining precise exceptions and program correctness. The front-end can support fetch from two threads, the second of which is dynamically spawned by the hardware rather than being statically created by the program. Initially, the only thread to run is the main (*primary*) program. The secondary (*future*) thread consists only of a program counter and register state. Out of the available rename registers, we dynamically reserve a certain number for the *future* thread, according to the program's current needs to exploit far-flung ILP. Once the *primary* thread runs out of its allocated registers, it stalls, and the *future* thread gets triggered and starts off from where the *primary* left off. This *future* thread cannot change the program state, *i.e.*, it cannot write to memory or update the *primary* thread's registers. It uses the remaining registers to rename and dispatch its instructions.

In order to allow the *future* thread to make progress beyond the instructions to which these registers are allocated, we relax the constraints on when its registers are released back into the free list. First, a register is released as soon as all its consumers

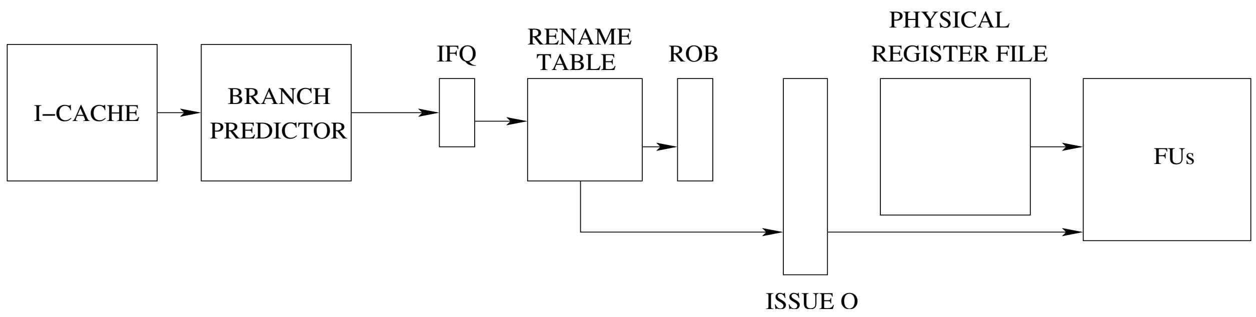

Figure 1. The base processor structure

have read its value, *i.e.*, we make the optimistic assumption that there will be no branch mispredicts or exceptions raised. The *future* thread cannot change the state of the *primary* thread — it serves the purpose of potentially warming up the register file, data and instruction caches, and resolving mispredicted branches early. Second, in order to avoid consuming *future* thread resources that prevent other independent instructions from executing, we also add a timeout mechanism to remove instructions that wait for operands in the issue queue for too long. This frees up registers and issue queue slots so that other productive dependence chains can make progress, thereby allowing the *future* thread to get far ahead of the *primary*. When the *primary* thread ceases to be stalled, it dispatches its subsequent instructions all over again, but makes speedier progress as its loads have been prefetched and its branches have been correctly predicted. The use of an Instruction Reuse Buffer (IRB) [29] could speed up the execution even more as some of these instructions would not have to be re-executed.

Thus, we rob the main program thread of some of its resources and allocate them to this opportunistic 'helper' thread that seeks independent instructions that are more distant. The benefit of such an approach would depend on the nature of the program, and we present a mechanism that dynamically performs this allocation of resources between the *primary* and *future* threads. As a result, in situations where the *future* thread degrades performance, the processor can always revert back to an organization like the base case, where all resources belong to the *primary* thread. Our simulation results indicate that relative to the base simulated architecture, performance is improved by an average of 17% with the dynamic helper thread.

The rest of this paper is organized as follows. We start by describing the proposed architecture in more detail in Section 2. In Section 3, we quantitatively evaluate its performance. This is followed by a discussion on the additional hardware requirements of the architecture in Section 4. Section 5 discusses related work, and we conclude in Section 6.

2 Proposed Microarchitecture

2.1 The Base Processor

In a typical processor (outlined in Figure 1), the processor front-end performs branch prediction, fetches instructions from the instruction or trace cache, and deposits them in the instruction fetch queue (IFQ). The IFQ holds the fetched instructions until they get renamed and dispatched into the issue queue. In the dispatch stage, the logical registers are mapped to the processor's pool of physical registers. The rename table keeps track of logical to physical register mappings and is used to rename instructions before putting them into the issue queue. The destination register is mapped to a new physical register that is picked out of the free list (the list of registers not presently in use). The mapping is also entered into the re-order buffer (ROB), which keeps track of register mappings for all instructions that have been dispatched, but not committed. The issue queue checks for register dependences and also has a store queue that ensures that loads are issued only when there can be no conflict from an earlier store. As instructions become ready and issue, they free up their issue queue entry. A branch stack within the rename table checkpoints the mappings at every branch, so they can be reinstated in the event of a branch misprediction. The structure just described closely resembles the R10000 [35] and the Alpha 21264 [14].

Instructions are issued from the issue queue when their register and memory dependences are satisfied, and they are committed from the ROB in program order as they complete. Consider the following example:

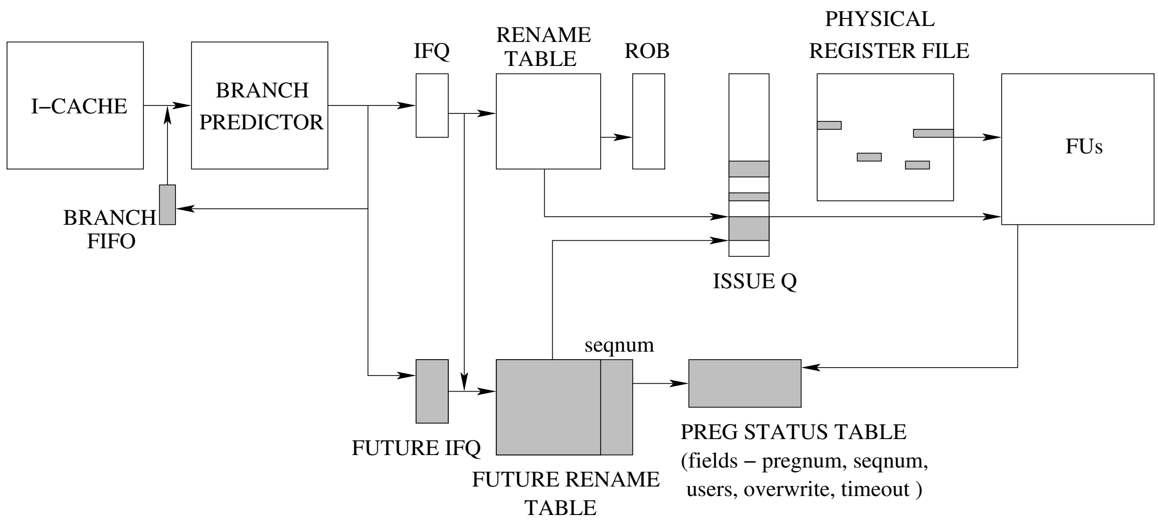

Figure 2. The architecture supporting the *future* thread (components belonging to the *future* thread are shaded).

```
Original code          Renamed code
lr7 <- ...             pr15 <- ...
... <- lr7             ...  <- pr15
branch to x            branch to x
lr9 <- lr3             pr31 <- pr19
lr7 <- ...             pr43 <- ...
...                    ...
end                    end
x:                     x:
... <- lr7             ... <- pr15
```

At dispatch, the first write to logical register 7 (lr7) causes it to get mapped to physical register 15 (pr15). This is followed by an instruction that reads lr7. The branch is then predicted to be not taken and the next instructions to be dispatched are a write to lr9 and a write to lr7. At this point, lr7 gets mapped to pr43 and subsequent users of lr7 will now read from pr43. Even if the instruction that reads pr15 has completed, pr15 cannot be released back into the free list unless the write to pr43 has committed. There are two reasons for this: (i) if the write to pr31 raises an exception, to reflect an accurate register file state, lr7 should show the value held in pr15, (ii) if the branch was mispredicted, we would need to jump to x, where the read from lr7 would actually refer to pr15. Hence, pr15 remains live until all instructions prior to the write to pr43 are known to not raise an exception and have all their branches resolved.

In the example shown above, if the write to pr31 was a load instruction that missed in the L2 and had to go to memory, it could occupy the head of the ROB for potentially a hundred cycles. If the processor has 24 rename registers, only up to 23 more instructions that write to registers can be dispatched in this period. This severely limits the ability of the processor to extract ILP.

2.2 Adding the Future Thread

The goal of the proposed architecture is to circumvent the in-order commit process in order to exploit any potential far-flung ILP in addition to nearby ILP. We begin with an overview of the proposed microarchitecture, followed by a more detailed description of the various operations.

As an illustrative example, we begin with a base processor that has 32 int and 32 fp logical registers, and 72 int and 72 fp physical registers (*i.e.*, there are 40 int and 40 fp rename registers). In the *future* thread architecture, the front-end, comprising the IFQ and the register rename table, is replicated (Figure 2). While the *primary* thread is not stalled, the *future* thread does not dispatch instructions, but it updates its rename table to reflect the new mappings in the *primary* thread. Of the 40 integer rename registers, 12 (for example) are reserved for the *future* instructions. When the primary thread runs out of registers and stalls, the *future* thread continues to make progress. It uses its allocated physical registers to dispatch subsequent instructions. These registers are then freed according to two criteria. Registers are reused as soon as there is no use for them (assuming no mispredicts and exceptions). In addition, if an instruction waits too long in the issue queue, it gets timed out and its register is reused. Instructions waiting in the issue queue for this register are also removed. Application of these two criteria is possible because of the fact that the *primary* thread will re-execute these instructions in order to ensure in-order commit and program correctness. Thus, registers reserved for the *future* thread can be reused much more quickly, potentially allowing the thread to execute far ahead of the *primary*, enabling prefetching of data into the cache, early branch prediction, and value reuse. The *future* thread does not engage in any speculation apart from speculating across branches. It respects register and memory dependences while issuing instructions.

2.2.1 Additional hardware structures

The three main additional structures are the *future* IFQ, the *future* rename table, and the Preg Status Table.

There are two program counters, one for the *primary* thread, and one for the *future*. These are identical at first, and fetched instructions are placed in each IFQs. Every cycle, instructions can potentially be renamed by both threads and dispatched into the issue queue. If the same instruction is being handled by both threads, the *future* thread will not dispatch it. The mapping corresponding to that instruction in the *primary* rename table is copied into the *future* rename table.

Each dynamic instruction is assigned a sequence number (this is a counter that wraps around when full and is large enough to ensure that all in-flight sequence numbers are unique — possibly 10 bits long). Sequence numbers are rolled back on a branch mispredict. These sequence numbers make it possible to relate the *primary* instructions to their *future* counterparts.

When the *primary* thread runs out of physical registers, it stalls. The *future* thread continues, using the remaining physical registers to map subsequent instructions. For each instruction that is dispatched by the *future* thread, an entry is added to the Preg Status Table. This is a small CAM structure, the size of the number of registers reserved for the *future* thread (12 entries, in this example, for int and fp each), that keeps track of the current physical registers in use within the *future* thread. The other fields in this structure are: (i) *Seqnum*, the sequence number corresponding to the instruction that has the physical register as destination, (ii) *Users*, indicating how many more consumers of that register still remain in the pipeline, (iii) *Overwrite*, indicating that the corresponding logical register has been remapped by a subsequent instruction, (iv) *Written*, indicating that the result has been written into the physical register, (v) *Timeout*, set to a particular value (30 in our case) at the time of dispatch, and decremented every cycle if the instruction has still not been issued. The *Users* field is incremented every time an instruction is dispatched that sources that physical register. It is correspondingly decremented when that instruction issues. The design considerations of the Preg Status Table are discussed in Section 4.

2.2.2 Timeout and register reuse

To help the *future* thread use its register resources more efficiently, we eagerly free up registers using the timeout mechanism and the register reuse criteria.

The rationale for the timeout mechanism can be illustrated by Figure 3. It shows a histogram of the number of instructions that wait in the issue queue for a given period of time. The particular example is that of a 20 million instruction window from the program *perimeter*, and is typical of most memory-intensive programs. It can be seen that instructions are made ready within the first few cycles of their dispatch, or after about 20 cycles, or after about 100 cycles. These correspond roughly to the L1, L2, and memory access times. The timeout heuristic models the fact that the non-readiness of an instruction in the first 30 cycles implies that it is waiting on a memory access and is likely to not be woken up for another 70 cycles. Hence, we time it out and allow its register and issue queue entry to be used by other instructions.

Registers get put back into the free list as soon as their overwrite and written bits are set and the number of users becomes zero. Likewise, when the timeout counter becomes zero, the register is put back in the free list, its mappings in the rename table (if still active) and the Preg Status Table are removed, and the instruction is removed from the issue queue. In order to ensure the correct execution of instructions, in the next cycle, the tag of this timed out register is broadcast through the issue queue and all instructions that source it, time themselves out. This not only frees up the issue queue slot but also ensures that the instructions do not wake themselves up when the same register tag (corresponding to the completion of a later instruction) is broadcast as ready. The process is repeated for the newly timed out instructions. *Future* instructions dependent on this value will not be dispatched due to the invalid entry in the rename table. This operation could take a few cycles depending

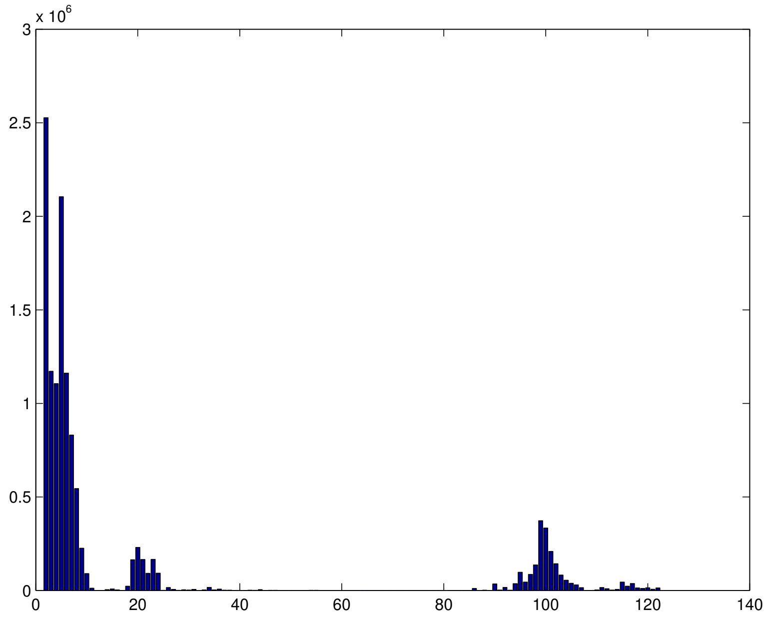

Figure 3. Histogram showing waiting time in the issue queue for a portion of the program *perimeter*. The X axis shows the time spent waiting in the issue queue, and the Y axis shows the number of instructions that waited for that period.

on the length of the dependence chain in the issue queue. To reduce hardware overhead, we could impose the restriction that *future* instructions only occupy certain issue queue slots, thereby having this associative logic for only a subset of the issue queue. While dispatching a *primary* instruction, if the issue queue is full, one of the *future* instructions is explicitly timed out to make room for it. This 'stealing' of issue queue slots ensures that priority is always given to the *primary* instructions.

2.2.3 Redispatching an instruction in the *primary*

When the instruction at the head of the ROB completes, the *primary* thread can start making progress again as registers get put in the free list. Instructions are fetched again from the I-cache into the IFQ and then dispatched. While dispatching an instruction, the Preg Status Table and *future* rename table are looked up. The *future* rename table keeps track of the sequence number for the last instruction that mapped the logical register within the *future* thread, while the Preg Status Table includes the sequence number of the instruction writing the physical register. The current instruction's sequence number is used to associatively look up the Preg Status Table. If a physical register mapping still exists for that instruction in the *future* thread, the same physical register is used to map the instruction in the *primary* as well. The corresponding physical register entry is removed from the Preg Status Table, as the register is no longer subject to the rules of the *future* thread. The *future* instructions that source this register need not update their operand tags. Also, the instruction need not be dispatched again into the issue queue, as the earlier dispatch will suffice to produce a result in that physical register. If a result already exists in the physical register, the *future* thread helps speed up the *primary* thread even more. This phenomenon is referred to as *natural reuse*. If a physical register mapping for that instruction does not exist in the Preg Status Table (the register has already been timed out or reused) and if there is a match with the sequence number associated with the *future* rename table's logical register entry, the *future* rename table is updated to reflect the mapping in the *primary* table.

2.2.4 Recovery after a branch mispredict

Once triggered, only the *future* thread accesses the branch predictor. It communicates its predictions to the *primary* thread through a FIFO queue. These predictions in the queue are updated when resolved by the *future* thread, so that the *primary* thread need not go along the mispredicted path.

When the *future* thread detects a mispredict, it checkpoints back to the state at the mispredict. However, some values may be lost (as the register might have been reused), thereby disallowing dispatch of instructions along some dependence chains.

As mentioned, the *future* rename table tracks the sequence number corresponding to the logical register mapping. A conventional rename table checkpoints its mapping at every branch. For the *future* thread, the mappings that might have been true at the time of checkpointing need not be true when the checkpoint is reinstated – instructions prior to the branch may have timed out, had their registers reused, or been re-dispatched as part of the *primary* thread. Hence, instead of checkpointing the mapping, we checkpoint the sequence number for the mapping. In addition, the Preg Status Table also checkpoints its overwrite bit. While reinstating the checkpoint, the sequence number is inspected to figure out where the correct mapping can be found. If the sequence number is less than the last sequence number encountered by the *primary* thread, then it means that the *primary* rename table has the correct mapping for that register. If the sequence number is greater, it means that the register, if still valid, should be part of the *future* thread and have a mapping in the Preg Status Table. In the subsequent cycles, these mappings are copied back into the *future* rename table so that it reflects an accurate state, and the overwrite bit is recovered. In Section 4, we suggest a few optimizations that would allow this process to be performed quickly and be overlapped with the time taken to fetch instructions from the correct path.

If the *primary* thread detects a mispredict, the *future* thread starts from scratch after copying the contents of the *primary* rename table.

2.2.5 Exploiting the IRB

In the microarchitecture described thus far, instructions may get executed by both the *primary* and *future* threads. An instruction reuse buffer (IRB) could be used to minimize this redundancy[1]. An implementation scheme like S_n or S_{n+d} [29] could be easily used with minimal modification. In our simulations, we use the S_n scheme because of its simplicity. In this scheme, the reuse buffer keeps track of the program counter, the operand names (register addresses) for an instruction, and the result value it produced when it was last invoked. During dispatch, if a program counter match is found in the IRB and the result value is valid, an instruction can bypass the issue and execute stages of the pipeline. Each instruction creates an entry in the IRB at the time of dispatch, and updates the result value at the time of completion. When an instruction dispatches, it also invalidates all the entries in the IRB that source the same logical register as its destination. Similarly, a store invalidates all loads in the IRB that have the same source address.

To support the *future* thread, two modifications need to be made to the IRB. *Primary* instructions cannot create IRB entries once the *future* thread is triggered (these entries may be invalid because the *future* thread may have dispatched instructions that have modified the operands, which the *primary* has no way of knowing). In addition, the entries in the IRB also keep track of the sequence number for the *future* instruction that produced them. The *primary* thread can reuse valid results in the IRB as long as these results were produced by instructions with sequence numbers smaller than or equal to that of the instruction being dispatched. This ensures that the contents of the logical registers that are the operands is the same as that used to generate the result.

2.2.6 Dynamic partitioning of registers

The allocation of physical registers between the *primary* and *future* threads need not be set at design time. In fact, a number of programs that do not have distant ILP would be better off using their registers to exploit nearby ILP rather than have the *future* thread throw those results away to advance further. We include a mechanism that dynamically accomplishes this partitioning on the fly. The number of registers allocated to each thread is controlled by stalling the thread's dispatch as soon as it has consumed its allotted registers. A simple counter keeps track of the registers allotted to and freed by each thread. A register, that can be dynamically set, specifies the maximum allowed counter value.

We use a simple interval-based mechanism [3] that monitors the program over regular intervals to decide what configuration to use in the next interval. After every 100K instruction interval, we examine a set of hardware counters that track the number of branches and the number of L1 cache misses. If there is a significant change in either of these compared to those in the last interval, we assume a change in program phase. Every new program phase is accompanied by an exploration process. For the subsequent intervals, the program is run with various register partitions, and the IPC for each interval is recorded. For example, if the processor has a total of 40 rename registers, we allocate eight to 40 registers to the *primary* thread, in steps of four. Thus, the exploration process takes nine 100K instruction intervals to finish. At the end of this exploration process,

[1] An IRB in a conventional microarchitecture exploits value locality by not re-executing instructions if they have the same operand values.

Fetch queue size	16
Branch predictor	comb. of bimodal and 2-level gshare; bimodal size 2048; Level1 1024 entries, history 10; Level2 4096 entries (global) Combining predictor size 1024; RAS size 32; BTB 2048 sets, 2-way
Branch mispredict penalty	9 cycles
Fetch, dispatch, issue, and commit width	4
Issue queue size	20 (int), 15 (fp)
L1 I and D-cache	64KB 2-way, 2 cycles, 32-byte line sizes
L2 unified cache	1.5MB 6-way, 15 cycles, 64-byte line size
TLB	128 entries, 8KB page size
Memory latency	70 cycles for the first chunk
Memory ports	2 (interleaved)
Integer ALUs/mult-div; FP ALUs/mult-div	4/2; 2/1

Table 1. Simplescalar simulator parameters

Benchmark	Input dataset	Simulation window	IPC of the base case
em3d (Olden)	20000 nodes, arity 20	500M-525M instructions	0.51
mst (Olden)	256 nodes	9M-14M instructions	0.44
perimeter (Olden)	32Kx32K	1515-1540M instrs	0.39
art (SPEC2k)	ref	500M-550M instrs	0.96
swim (SPEC2k)	ref	1000M-1025M instrs	0.73
lucas (SPEC2k)	ref	2000M-2050M instrs	1.03
sp (NAS)	A, uniprocessor	2500M-2550M	0.98
bt (NAS)	A, uniprocessor	3200M-3250M	0.71
go (SPEC95)	ref	1000M-1025M	1.29
compress (SPEC95)	ref	2000M-2025M	1.53

Table 2. Benchmark description

the partition that worked best is used until the next phase change is detected. This process of recording IPCs and picking the best configuration is easily done in hardware with simple logic, or in software by low-overhead interrupt handlers (like that used for software TLB refill). Some programs do not show consistent behavior across 100K instruction intervals and spend most of their time in the exploration phase. To detect such a scenario, we maintain a counter that is incremented by 1 for every interval in stable state and is decremented by 10 every time a phase change is detected. If the counter value falls below a certain threshold, we shut off the exploration process and resort to the register partitioning that was picked most frequently.

The timeout mechanism and recovery after a branch mispredict result in lost values that cripple some dependence chains. This could lead to some noise in the IPC measurements during the exploration process. Hence, if we notice that the IPC during the stable state is often significantly different from the IPC observed for that partition during the exploration phase, we declare a phase change and repeat the exploration.

3 Results

3.1 Methodology

We used Simplescalar-3.0 [4] for the Alpha AXP instruction set to simulate a dynamically scheduled 4-wide superscalar processor. The simulation parameters are summarized in Table 1.

The simulator has been modified to model the memory hierarchy in great detail (including interleaved access, bus and port contention, writeback buffers). We also model a physical register file and an issue queue that is smaller than the ROB size. (In Simplescalar, the issue queues and the ROB constitute one single unified structure called the Register Update Unit (RUU).) These are further divided into separate integer and floating-point structures.

Our base processor has parameters resembling the Alpha 21264 [14]. We use 72 integer[2] (int) and 72 floating-point (fp) physical registers (corresponding to 40 rename registers, int and fp, each) and integer and fp issue queues of 20 and 15 entries, respectively. We use a sufficiently large ROB as it is a relatively simple structure and is likely to not be on the critical

[2]The Alpha has 80 integer registers. We use 72 for uniformity.

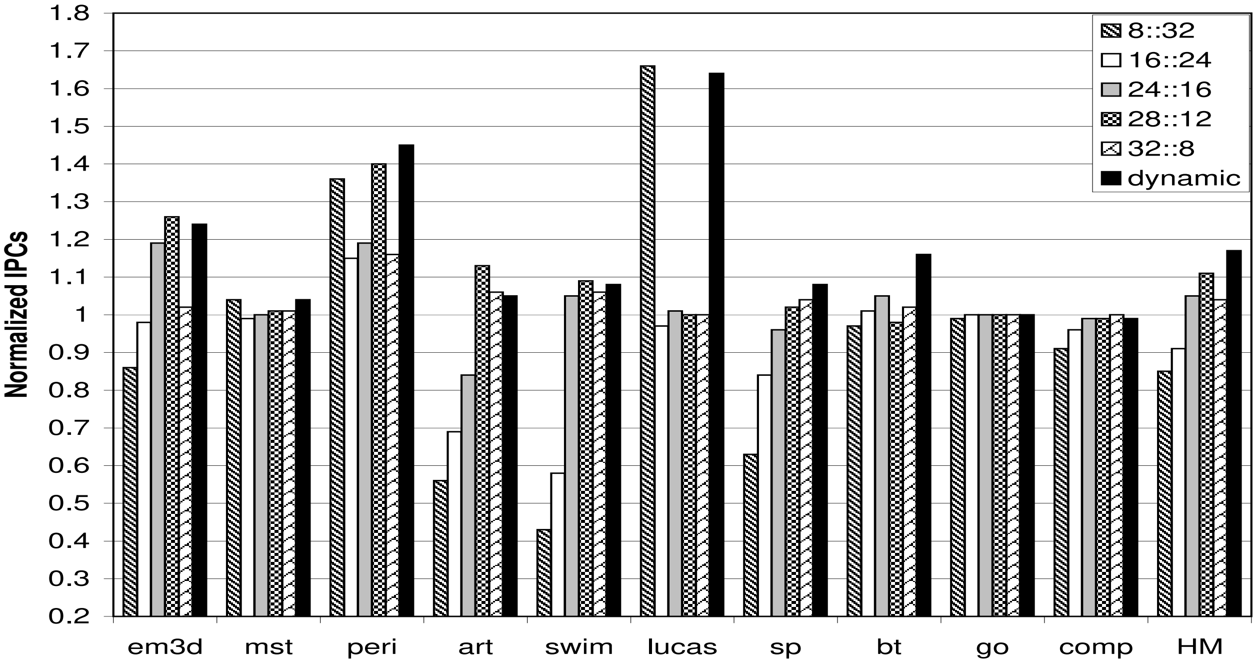

Figure 4. Performance of the *future* thread for various fixed register allocations between the *primary* and *future* thread. For example, '8::32' represents an allocation where 8 rename registers are reserved for the *primary* thread and the remaining 32 are reserved for the *future*. The last bar shows performance with the interval-based scheme that dynamically picks the best allocation. IPCs have been normalized with respect to a base case that has no *future* thread and uses all 40 rename registers for the *primary*.

path. Dispatch gets stalled as soon as either the registers or the issue queue entries get used up, so the ROB occupancy rarely exceeds 80 entries, which is the ROB size in the 21264. Our goal is to demonstrate potential improvements on an existing processor model. In addition, we present results with and without a small 16-entry fully-associative IRB with the S_n implementation scheme.

We ran our simulations on 10 programs from SPEC2000, SPEC95, the NAS Parallel Benchmark [8], and the Olden suite [23]. Eight of these programs are memory-intensive and suffer the most from the problem of a single long latency instruction holding up the commit stage. We have also included two non-memory-intensive programs (*go, compress*) from SPEC95 INT, to illustrate the effect of the *future* thread on this class of applications. To reduce simulation time, we studied cache miss rate traces to identify program warm-up phases and smaller instruction windows that were representative of the program behavior[3]. The programs were also run for 1M instructions in detail to warm up the various structures before measuring performance. Details on the benchmark are listed in Table 2. The programs were compiled with Compaq's cc, f77, and f90 compilers for the Alpha 21164 at the highest optimization level.

3.2 Analysis

We start by showing the performance with a *future* thread when there is a fixed allocation of registers between the *primary* and *future* threads. This motivates the use of the interval-based scheme to do a dynamic allocation, which we then use throughout the rest of the paper. The improvement is first analyzed and attributed to the various components of the *future* thread. We then look at the effect of various parameters like the IRB, issue queue, and register file size. Finally, we also evaluate future processor models with projected parameters.

[3]Because each iteration in *bt* is very long, we used a smaller window than was representative of the whole program. However, the results were selectively verified to be indicative of the performance over longer windows.

	em3d	mst	peri	art	swim	lucas	sp	bt	go	comp
Num timeouts	0.29	1.12	0.56	0.31	0.42	0.59	0.37	0.16	0.00	0.03
Num eager reg release	0.45	0.03	0.65	0.30	0.11	0.06	0.13	0.28	0.01	0.06
Num natural reuse	0.14	0.13	0.20	0.23	0.37	0.25	0.22	0.26	0.10	0.16
Avg dist between oldest and youngest instrs (base, *future*)	71, 136	25, 115	51, 114	63, 131	67, 123	31, 183	75, 128	47, 75	19, 19	39, 49
Num loads issued by *primary* thread that take more than 40 cycles (base, *future*)	0.12, 0.05	0.02, 0.02	0.11, 0.05	0.02, 0.01	0.04, 0.04	0.05, 0	0.03, 0.02	0.05, 0.04	0, 0	0, 0
Num *future* instrs issued	0.7	0.2	1.4	0.8	0.8	0.6	0.6	0.9	0.2	0.4
Branch direction prediction rate (rounded off)	95%	97%	94%	98%	99%	98%	89%	98%	80%	93%
% of mispreds detected by *future* instrs	88%	0%	59%	42%	74%	99%	73%	68%	4%	3%
IRB hit rate for *primary* thread	20%	5%	10%	35%	8%	0%	5%	14%	22%	16%

Table 3. Various statistics pertaining to the *future* thread (with a dynamic allocation of registers) and the base case with no *future* thread (most numbers are normalized to the number of committed instructions, for example, Num timeouts is the number of timeouts per committed instruction).

3.2.1 Dynamic partitioning of registers

Figure 4 shows speedups with the *future* thread for various fixed allocations of registers between the *primary* and *future* threads. For all figures, the IPCs have been normalized with respect to an identical base case that has no *future* thread (*i.e.*, all rename registers are allocated to the *primary* thread). Of these various static organizations, the 28::12 allocation that reserves 28 registers for the *primary* thread has the best overall speedup (when comparing the harmonic mean (HM) of IPCs). However, we see that different allocations do well for different programs. This depends on whether the program has distant or nearby ILP and whether the number of registers reserved for the *future* thread are enough to allow it to advance far enough to exploit this distant ILP. The highest speedups for *lucas* and *mst* are seen by reserving only eight registers for the *primary* thread, but this is the worst allocation for a number of programs that also have nearby ILP. This motivates the need for a dynamic scheme that picks the right allocation on the fly, depending on program requirements. The last bar in Figure 4 shows that the overall speedup of 1.17 with the interval-based dynamic scheme far exceeds the speedup of 1.11 possible with the best static organization. The only program that experiences a number of phase changes is *art* as it does not have consistent behavior across 100K instruction intervals. Hence, after a number of initial exploration phases, it finally remains fixed at the organization that was picked most often. This organization need not be the best for the whole program and hence in a few experiments, we see performance degradations for *art* depending on which organization was finally picked. All subsequent discussions and results assume the use of the dynamic allocation of registers between the *primary* and *future* threads.

3.2.2 Effects of prefetch, branch resolution, and reuse

Table 3 shows various statistics that help us explain the behavior of the *future* thread. In Figures 5 and 6, we attempt to isolate the contributions of the various components to the performance of the *future* thread. In Figure 5, the first bar (prefetch only) shows a *future* thread implementation that just runs ahead along predicted paths to warm up the data and instruction caches, while ignoring the outcome of all branch instructions. In this scenario, branch mispredicts are discovered only when the *primary* thread re-executes the branch instruction. The second bar shows an implementation where the *future* thread also resolves branch mispredicts early and initiates recovery. The third bar represents a model that adds an IRB. We see that a significant portion of the improvement is due to the prefetch effect, with the overall speedup being 1.12. Table 3 shows that there is a sharp drop in the number of long latency loads seen by the *primary* thread. The number of loads per committed instruction that see a latency of more than 40 cycles falls by almost a factor of two and is even reduced to zero in the case of

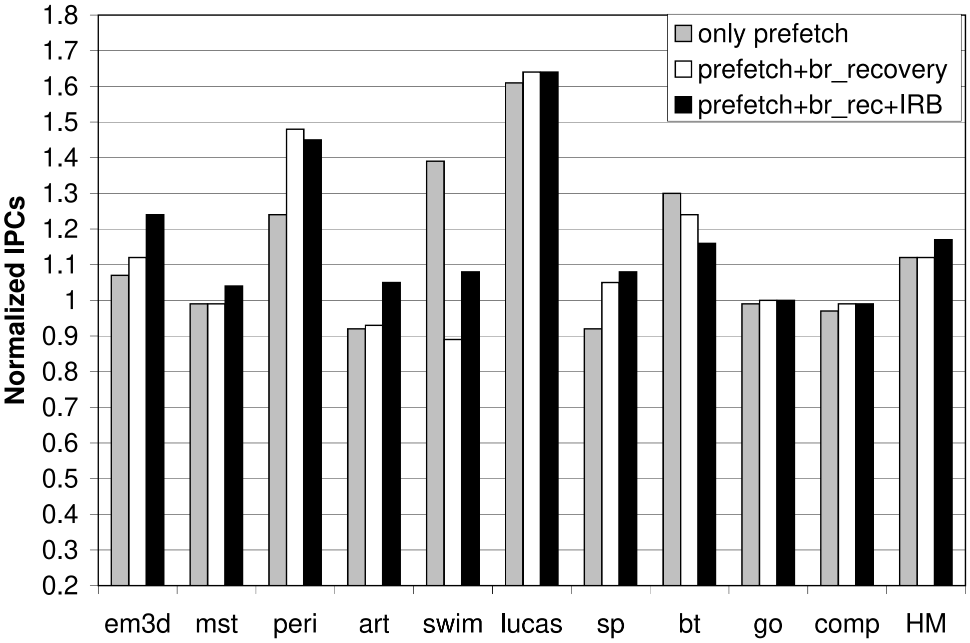

Figure 5. *Future* **thread performance broken down as prefetch, early branch recovery, and reuse.**

lucas. For *lucas*, the dynamic scheme allocates most rename registers to the *future* thread and this enables it to advance as far as the next loop iteration, thereby fetching the data long before the *primary* thread starts that iteration.

When the *future* thread is allowed to initiate early branch recovery, we see significant improvements for the programs with low branch prediction accuracies. This results in an additional improvement of 5%, 24%, and 13% in *em3d, perimeter*, and *sp*, respectively. On the other hand, we see a big drop in performance for *swim*. When the *future* thread initiates early branch recovery, it tries to restore a valid register state. Because of the eager release of registers, some values remain lost, disallowing progress along those dependence chains. This sets off a chain reaction, where the *future* thread runs much further ahead but is unable to execute any of the instructions. It can be productive again only when the *primary* thread catches up, which occurs when the *primary* discovers a branch mispredict (for a branch not executed by the *future*) and squashes all subsequent instructions. *Swim* is a loop-based floating-point code and has a low branch mispredict rate. As a result, the *future* thread may have to wait a very long time before it has valid register mappings. This effect is also somewhat seen for *bt*. This negative effect of early branch recovery can be easily eliminated by not attempting it for programs with high branch prediction accuracies. Our simulations do not assume the use of such a scheme.

Finally, by adding the IRB we see an additional overall improvement of 5%. A number of instructions that have been dispatched by the *future* thread need not be re-executed when seen by the *primary* thread. The last row in Table 3 shows that up to 35% of these instructions can obtain their result from the IRB. This IRB hit rate improves slightly when we use larger IRBs. Using a 128-entry IRB, we see additional improvements of 8% and 7% in *mst* and *bt*, resulting in an additional 1% overall improvement.

3.2.3 Breakdown of contributions

Three major design components enable the *future* thread to advance ahead of the *primary*. From Table 3, it can be seen that the average distance between the oldest and youngest instruction within the processor increases greatly because of the *future* thread. This number represents the size of the in-flight instruction window. The largest window seen by the base architecture is only 75 instructions (in the case of *sp*), but the *future* thread can look in a much larger window (as large as 183 in the case of *lucas*) because of the eager release of registers and the timeout mechanism. Both of these often come into play as evidenced by the statistics in the first two rows of Table 3. In addition, Table 3 demonstrates that a significant number of instructions need not be re-executed by the *primary* thread if their mapping still exists in the *future*, which we describe as natural reuse.

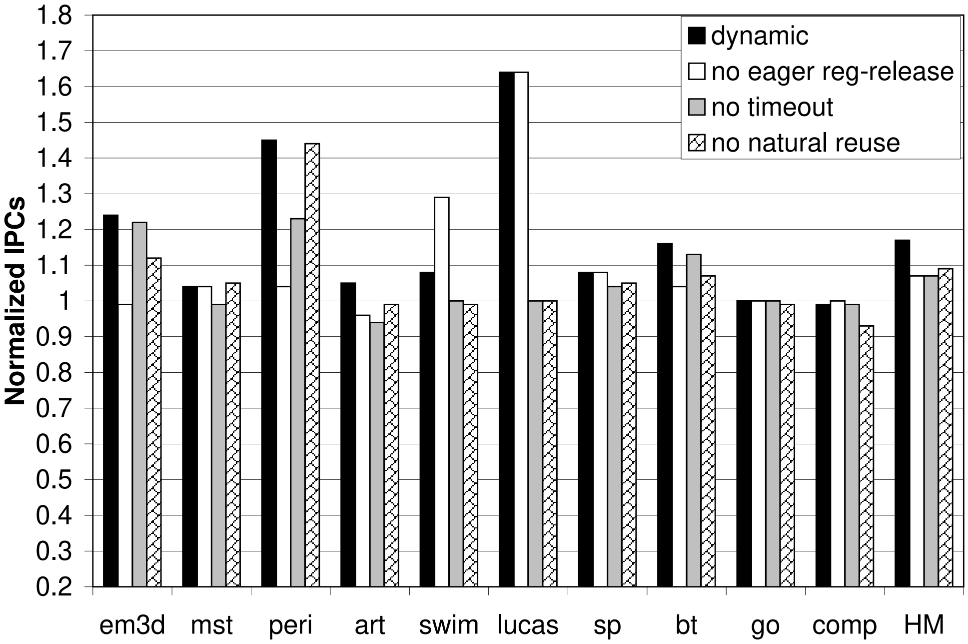

Figure 6. The contributions of the various features of the *future* thread. The leftmost bar represents the dynamic scheme with all features turned on. The other three bars show performance as each feature is disabled.

Figure 6 quantifies the contributions of each of these three components by disabling them one at a time. It can be seen that eager register release accounts for most of the speedup in *em3d* and *perimeter*, while timeout helps greatly in *perimeter* and *lucas*. For *lucas*, the primary bottleneck is the issue queue. The use of the timeout mechanism makes it possible to reduce contention for the issue queue, thereby not stalling dispatch. Similarly, by allowing natural reuse, we prevent the re-dispatch of instructions into the issue queue, thus alleviating the bottleneck again. Thus, the combination of the timeout mechanism and the natural reuse allows the *future* thread to advance far enough to do an effective job prefetching. Eliminating any of these features results in overall speedups of only 1.07-1.09. Eliminating eager register release results in an improvement for *swim* because an early recovery from a branch mispredict by the *future* thread now results in no lost values, thereby eliminating the problem alluded to earlier.

We see almost no improvements for non-memory-intensive programs like *go* and *compress*[4] as they rarely run out of registers, thereby not triggering the *future* thread.

3.2.4 Effect of various processor parameters

Mst is a memory-intensive program that does not show much improvement as it has little nearby ILP, causing instructions to wait in the issue queue, thus stalling dispatch. For the other programs, by using the *future* thread, the register file is removed as the bottleneck to dispatch. Hence, stalls are usually caused by the small size of the issue queue. We next evaluate the *future* thread for a processor model that has larger int and fp issue queues of 30 entries each. The larger issue queues resulted in no improvement for the base case, but they enabled the *future* thread to advance even further, resulting in an overall speedup of 1.21 (see Figure 7).

We next study the effect of different register file sizes. Figure 8 shows speedups with the *future* thread for processor models that have physical register file sizes ranging from 56 to 80 registers (int and fp, each). Each bar uses the corresponding base case to compute speedups. Two effects come into play here. Using a smaller register file makes it more of a bottleneck, increasing the potential benefit of the *future* thread. However, with a smaller register file, the *future* thread will also be

[4] *Compress* has a high L1 miss rate, but a small L2 miss rate, and the instruction window in the base processor is large enough to hide L2 latencies.

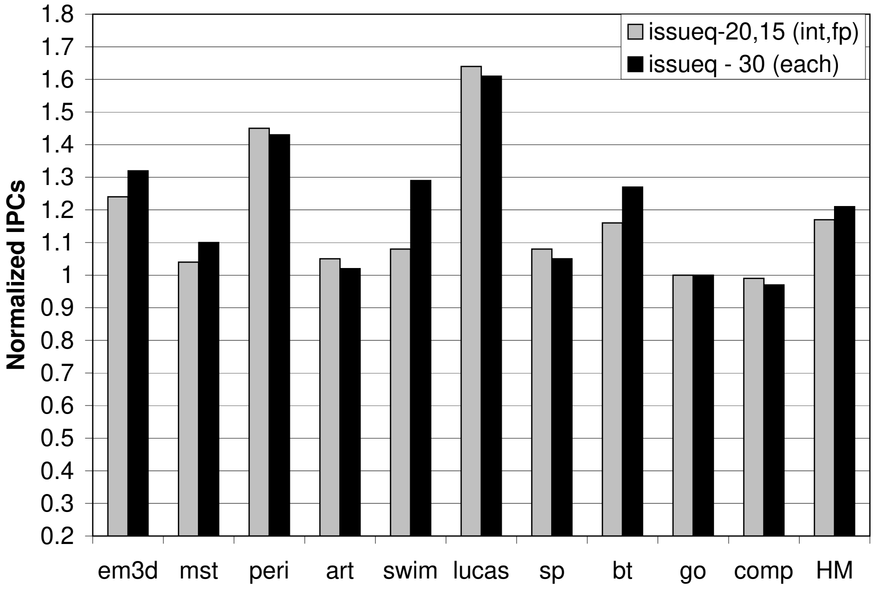

Figure 7. The effect of a larger issue queue. The left bar shows speedups with the *future* thread for the Alpha-like processor, while the right bar shows speedups for a processor model that has the same parameters except for a larger issue queue.

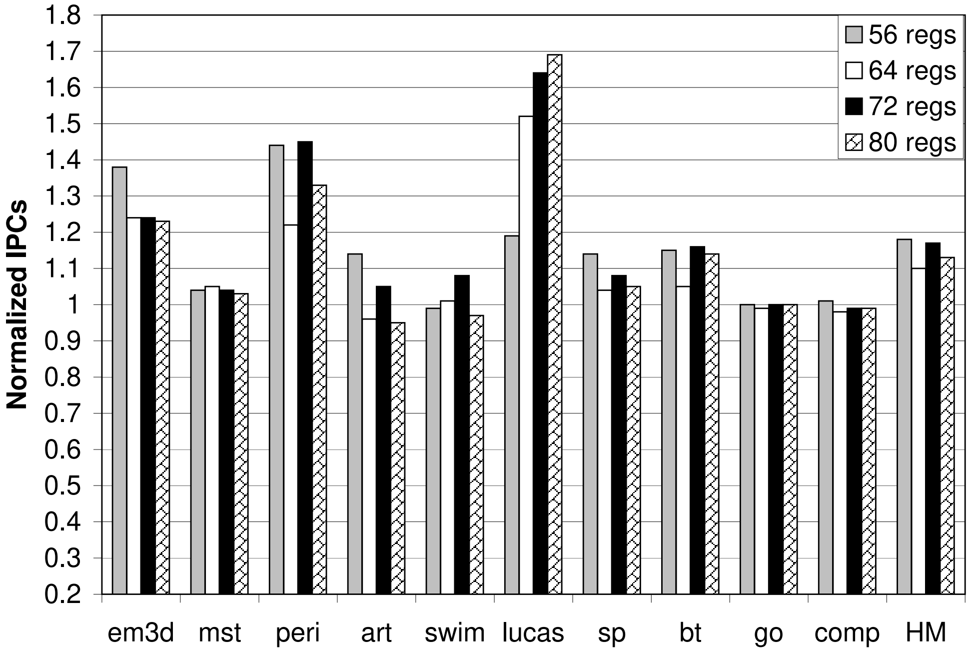

Figure 8. Speedups with the *future* thread for processor models that have different register file sizes.

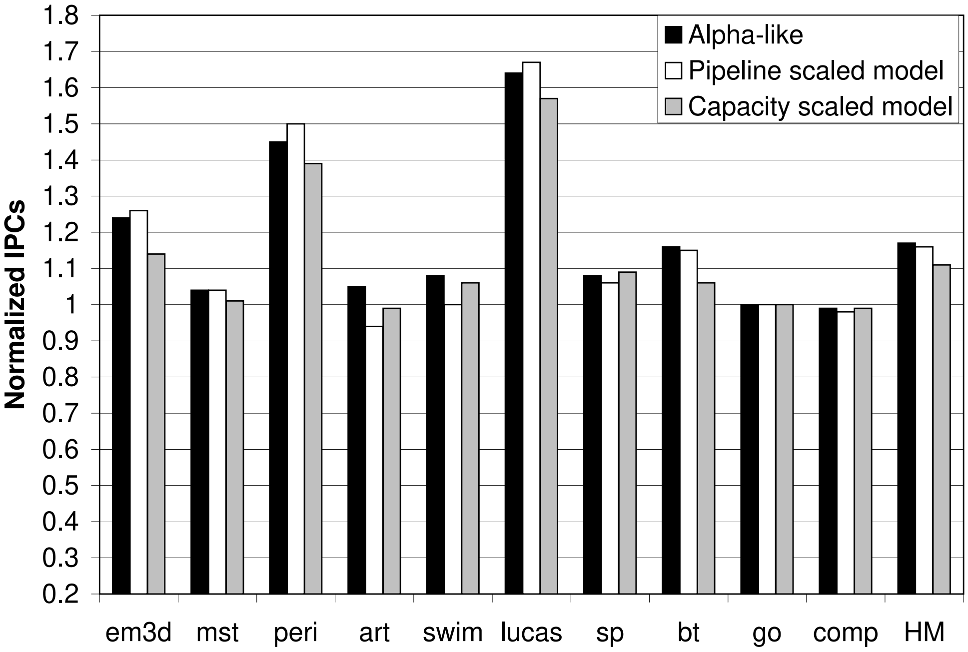

Figure 9. Speedups for two future processor models.

limited in its ability to look ahead, reducing the prefetch effect. Depending on which effect dominates, we see different behaviors for the different programs. Hence, a clear trend is not seen in the overall speedup numbers. It must be pointed out that the raw IPC for a 56-register base case augmented with the *future* thread (0.72 IPC) is better than the raw IPC for a 72-register base case without the *future* thread (0.71 IPC). While the IPCs are comparable, the former processor model is likely to have a faster clock speed.

The processor model that has been described and simulated assumes that both the *primary* and *future* threads can rename and dispatch up to four instructions each in every cycle. This would require as many as eight write ports in the issue queue. Further, we have assumed that additional paths exist in the issue queue so that tags of timed out instructions can be broadcast. We also simulated a processor model that takes into account these constraints. Only up to a total of four instructions from either the *primary* or *future* thread were allowed to dispatch. Only up to a total of four instructions were allowed to either issue or timeout every cycle. These two constraints ensure that we do not introduce additional complexity, as compared to the base case. Only *swim* and *sp* showed modest performance loss because of this, but they still outperformed the base case.

Finally, we evaluate two other projected future processor models. We use the approaches outlined by Agarwal et al [1] to define a pipeline-scaled model that keeps structure sizes the same but uses deeper pipelines and a capacity-scaled model that shrinks structure sizes so as to not have deep pipelines. The pipeline-scaled model increases the L1, L2, memory, and mispredict latencies to 3, 20, 105, and 15 cycles, respectively. The capacity-scaled model uses 64 physical registers, issue queue sizes of 15 (int) and 12 (fp), 32KB 2-cycle L1s, 1MB 15-cycle L2, memory latency of 105 cycles, and a mispredict penalty of 10 cycles. Figure 9 shows that the speedups go up slightly for most programs with the pipeline-scaled model because the longer memory access times further necessitate the use of a prefetch thread. No overall benefit is seen because there is a performance degradation for *swim* due to an even greater number of lost register values following early branch recoveries. Not as much benefit is seen for the capacity-scaled model. This is partly because the issue queue becomes a greater bottleneck and partly because the smaller register file does not allow the *future* thread to run much further ahead.

3.2.5 Combination with hardware prefetch

The *future* thread has the ability to prefetch data in codes with unpredictable control and data flow. A number of the programs in this study also have regular accesses and can be effectively handled by the use of a hardware stride predictor [6]. We next evaluate the effect of such a hardware prefetch mechanism on the behavior of the base processor model and the *future* thread.

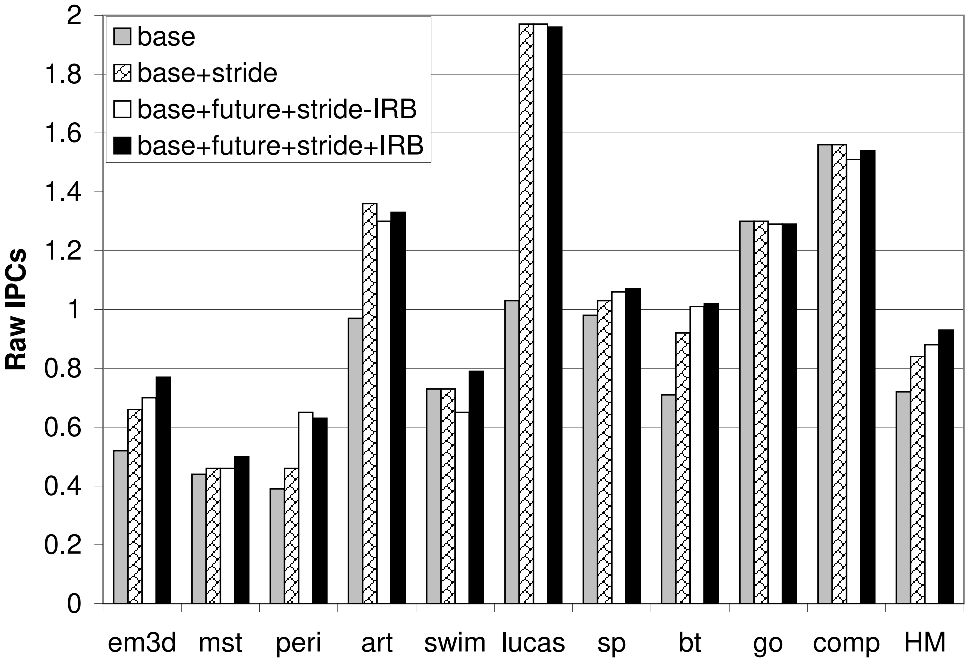

Figure 10. Raw IPCs for the base case, for the base case with a stride prefetcher, and for the combination of the *future* thread and the stride prefetcher (without and with an IRB).

Figure 10 shows raw IPC numbers for four models. The first bar shows the IPC of the base case and the second bar shows the IPC for a base case augmented with a stride prefetcher. In our simulations, we use a 512-entry direct-mapped stride prefetcher (indexed by program counter) that maintains a finite state machine for each of the most recently issued load instructions. The finite state machine determines if the data accesses are regular. When a load is encountered, it issues a prefetch for the next load if the last two or more loads had the same stride. Prefetches are issued only if the stride exceeds 16 bytes in order to avoid unnecessary prefetches due to the spatial locality within a cache line. Prefetches contend for cache ports just like any other load operation and bring data into the various levels of cache, *i.e.*, no separate prefetch buffer is used.

We see big improvements for a number of programs, resulting in an overall speedup of 1.17. We also see significant improvements for pointer-chasing applications like *em3d* and *perimeter*. The data access patterns in these applications do not change over time, and hence, following the pointers results in accesses to regularly strided addresses. Such speedups would probably not be seen for other pointer-intensive codes with dynamic data structures. We then employ the *future* thread in addition to having the stride prefetcher[5] (shown by the third bar in the figure). We do not use the IRB in this case so as to isolate only the effects of prefetch and early branch recovery. We notice reasonable improvements in a number of cases. Because the *future* thread jumps ahead and issues loads early, it also ends up issuing the prefetches early. This helps to hide the memory latency even beyond that which is possible with just the stride prefetcher. In the case of *lucas*, the stride prefetcher does a nearly perfect job, resulting in no additional speedups from the *future* thread. The overall speedup compared to the base case is 1.22, in spite of the slowdown seen for *swim* due to the loss of register values following a branch mispredict. The last bar also includes an IRB, thus showing the full benefit of the *future* thread. The overall speedup is now 1.29.

These results indicate that the *future* thread can be combined with other forms of prefetch to yield even greater speedups.

4 Additional Hardware Requirements

In this section, we qualitatively discuss the extra hardware required to implement the *future* thread.

[5]When triggered, only the *future* thread can update the finite state machine of the stride predictor. Both threads issue prefetches.

Future Rename Table

The rename table for the *future* thread constitutes the biggest overhead in terms of transistors. In addition to the register mapping, it also stores the sequence number. While a conventional rename table checkpoints the mapping in the branch stack, the *future* rename table checkpoints the sequence number. Each cell in a rename table contains a shift register to checkpoint values. Since this constitutes most of the cell, the size of the value being checkpointed plays a large part in determining the table's access time [21]. A conventional rename table checkpoints 7-bit values (the physical register tag), while the *future* rename table checkpoints the sequence number (a 9-10 bit value). While this implies a longer access time for the rename table, the results in Palacharla et al [21] indicate that the rename table is not on the critical path for the technology parameters examined.

In addition, while looking up the *primary* rename table, an access has to be made to the Preg Status Table to detect if a mapping for that instruction exists in the *future* thread. This lookup can be done in parallel with the lookup of the *primary* rename table and should not affect the critical path.

The operations to be performed by the *future* rename table on a branch mispredict are another source of complexity. The checkpointed sequence number has to be examined and based on this, the mapping has to be copied from either the Preg Status Table or from the *primary* rename table. Since all these structures have a limited number of read and write ports, copying as many as 64 mappings could take a number of cycles. One possible optimization would be to checkpoint the actual mapping instead of the sequence number when it is known that the mapping cannot change. For example, if the sequence number indicates that the instruction that set this mapping has been dispatched in the *primary* thread, then it is known that this mapping will still be true when the branch mispredict is discovered. Hence, in this case, by checkpointing the mapping, a copy need not be made at the time of mispredict recovery. Even with these optimizations, it is still possible that the recovery could add a few cycles to the mispredict penalty for the *future* thread. We simulated the effect of an extra four cycle mispredict penalty and noticed only a 1% performance degradation in two of the programs with high misprediction rates. Given the opportunistic nature of the *future* thread, its mispredict penalty does not play a very major role in affecting performance.

Preg Status Table

The Preg Status Table stores the sequence number, the physical register tag, the users counter, the overwrite bit, the written bit, and the timeout counter. The overwrite bit will have to be checkpointed at every branch, and would hence include a shift register in its cell. This amounts to a total overhead of not more than 25 bits per entry. While it has been logically described as one structure, it can be broken up into a number of small CAM structures, as each field can be independently accessed. The most complex of these would be the users field which would need as many as 16 ports (corresponding to two operands for each of four instructions being renamed and four instructions being issued). This structure would still be a lot smaller than a rename table that has as many ports, much larger fields per entry, and many more entries. Also, the counters could be approximated by a simpler structure that used fewer ports as it is unlikely that all possible 16 ports would be used in a cycle.

In addition to these two structures, some negligible overhead is imposed by the branch FIFO (a simple array of bits, typically not exceeding 50 bits), and the *future* IFQ that buffers instructions before they are dispatched into the issue queue. A counter and associated logic would be required for the sequence number. The dynamic scheme requires a few additional counters to monitor the frequency of branches and cache accesses (hardware profiling counters are already present in most modern processors). The IRB, as implemented in our simulations, is a fairly simple structure. The S_n implementation scheme does not affect other processor critical paths. An IRB with only 16 entries can eliminate much of the redundancy that results from the execution of the *future* thread.

5 Related Work

Dundas and Mudge [10] introduced a scheme for halting the *main* instruction stream on a cache miss, and running ahead to prefetch data. However, this scheme was only applicable to an in-order machine with no instruction-level parallelism support. In our method, because the base processor is a dynamic superscalar architecture, the *main* instruction stream is never halted - once it runs out of registers, we use spare registers as a scratchpad to make forward opportunistic progress. Thus, the implementation is vastly different, the performance benefits are far greater, and we are targeting cases where we run out of physical registers.

The idea of forming multiple threads that execute distant instructions has been exploited in a number of approaches, such as Multiscalar [30], Trace processors [25], DMT [2], and TLDS [31], that use hardware or compiler generated threads to fork off computation that might be far ahead of the current commit stage of the processor. These are all hardware intensive solutions as they assume that there would effectively be a separate processing unit or a Simultaneous Multithreaded (SMT [33]) base to execute these threads. They require significant hardware to store results and to transfer register values between threads to free up dependences. They are also highly speculative in nature, as these threads might lie much further ahead in the program control flow. The *future* thread is a lot less speculative as it starts off from where the *main* thread left off. Dependences are resolved just as in the baseline processor, without any added hardware (although we do complicate the dispatch stage). The *future* thread makes forward progress by relaxing the constraints on register release and by having the timeout mechanism, which are both unique to this approach.

Zilles and Sohi [36] characterize problem instructions (cache misses, branches) and the instructions that lead to them. They point out that a smaller subset of the program code can be pre-executed so that the *main* instruction stream does not encounter cache misses or branch mispredicts. They assume an underlying implementation that can pre-execute these slices. Roth and Sohi [28] talk about such an implementation that can pre-execute certain dependence chains (hence, referred to as data-driven multithreading). They use profiling to generate these slices and annotate the code to trigger them at appropriate points. These threads use physical registers to store their results and they are integrated into the main program thread when it catches up. Our approach is less concerned with the ability to fork off a previously created thread early. It is a brute-force method that tries to make forward progress even when the registers run out, and does not rely on compiler or profiling help. We also show how an IRB can be used to integrate results back into the *main* thread instead of keeping results around in registers.

There have also been a couple of attempts at improving branch resolution by pre-execution [11, 27], where the slice determining the branch is duplicated and made to run in a separate window (sharing the same physical registers). Farcy et al [11] notice regularity in the branch condition computations and use value prediction to accelerate the second thread.

Simultaneous Subordinate Microthreading (SSMT) [5] and Assisted Execution [9] are similar schemes where custom-generated threads are invoked within the hardware by certain events. These threads do very simple specific things and cannot be automatically generated.

A related concept is AR-SMT [24] and SRT [22], which run two copies of the same program on an SMT processor and compare results from both threads. Their goal is to detect transient faults in a chip, rather than to enhance performance. An extension of this is the Slipstream processor [32], where the thread running ahead is a shortened version of the original program (dynamically created by detecting and eliminating ineffectual pieces of the program), and the trailing thread is the full program that verifies the correct working of the leading thread. The two programs together can run faster than the single original program because the leading thread communicates values and branch outcomes to the trailing thread as (often correct) predictions.

There are ongoing efforts to use the register file in a complexity-effective manner. Cruz et al [7] present a multi-banked register file, with the different banks having different speeds. While this degrades IPC, it enables a faster clock, thus improving overall execution time. Other work [17, 34] proposes improving register utilization by allocating registers when instructions complete. The relaxed conditions for releasing registers into the free list have been proposed before [18] in the context of processors with imprecise exceptions.

The primary advantage of the *future* thread is its prefetching effect. A number of hardware [6, 13, 26] and software prefetching [16, 19] schemes have been proposed. Most of these schemes can do a better job of prefetching as they exploit some higher-level program information (regularity of accesses). This regularity can be determined at compile time or as strides or load-value dependences in hardware. This lack of high-level information prevents us from doing a very effective job of prefetching. We, however, do a more exact job as we respect dependences and actually compute load addresses (rather than use heuristics like most hardware prefetch schemes). We also use dynamic branch prediction to follow the probable control-flow path, instead of greedily prefetching [16] along all possible paths. This prevents us from fetching useless lines into the cache (unless we are on the wrong branch path). Hence, our techniques are also applicable to irregular codes with unpredictable control flow and unpredictable data accesses. Luk [15] addresses a similar problem in the context of an SMT processor by using the compiler to help pre-execute these codes. We do not add instruction overhead, unlike software prefetching schemes. Some of the prefetch schemes can also be combined with the *future* thread to yield greater speedups, as demonstrated in Section 3.

There has also been a software approach to tackling the problem of a single cache miss holding up the ROB. Pai and Adve [20] describe a compiler algorithm that restructures code so that cache misses are clustered, thereby increasing the memory parallelism while the ROB is stalled. Compiler based methods can only be targeted at regular codes. The *future* thread can jump ahead and issue *future* memory operations while the *main* instruction window is stalled, thus achieving the same effect of improved memory parallelism.

To our knowledge, an Instruction Reuse Buffer [29] has only been applied to exploit value locality and squash reuse (reuse of instructions that get squashed because of a mispredict, but lie on both paths out of the branch). Our proposal is a novel application of the IRB.

6 Conclusions and Future Work

We have designed and evaluated a microarchitecture that dynamically allocates a portion of the processor's physical resources to a *future* thread in order to exploit distant ILP in addition to nearby ILP. Long latency instructions tend to stall the commit phase of a traditional superscalar architecture on reaching the head of the re-order buffer. Subsequent instructions use up the available physical registers, after which the dispatch stage stalls. In our proposed microarchitecture, part of the physical registers are allocated for the *main* program and once they are consumed, the *future* thread gets triggered and makes forward progress. It eagerly releases registers and times out instructions that wait too long in order to opportunistically advance far beyond what the *primary* thread is capable of. It thus improves performance by resolving branch mispredicts early, by warming up the data cache, the instruction cache, and the instruction reuse buffer, and by reusing register mappings and values. In addition, an interval-based scheme is used to allocate the optimal number of registers to the *future* thread.

Our evaluation on some of the more memory-intensive benchmarks show very promising speedups of up to 1.64. The overall improvement on our benchmark suite is 17%. The contributions come mainly from prefetching, with significant contributions from early branch recovery in the programs limited by poor branch prediction accuracies. The use of a small 16-entry IRB accounts for 5% of this improvement. The dynamic allocation of registers plays a major role in tuning the hardware to the ILP requirements of each program phase. The use of a larger issue queue allows the *future* thread to achieve an overall speedup of 1.21. We also observed impressive speedups for a variety of processor models and even in combination with a hardware stride prefetcher.

This work has only realized some of the potential that such an organization promises. Given that the *future* instructions do not change processor state and only serve to warm up the cache and the reuse buffer, we can relax some of the hard constraints that are ordinarily imposed on the main processor. The *future* thread could engage in aggressive memory dependence speculation and value prediction to advance even further. In addition, compiler support could be used mark a subset of instructions that the *future* thread could restrict itself to.

References

[1] V. Agarwal, M. Hrishikesh, S. Keckler, and D. B. urger. Clock Rate versus IPC: The End of the Road for Conventional Micr oarchitectures. In *Proceedings of the 27th ISCA*, pages 248–259, 2000.

[2] H. Akkary and M. Driscoll. A Dynamic Multithreading Processor. In *Proceedings of MICRO-31*, pages 226–236, 1998.

[3] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures. In *Proceedings of MICRO-33*, pages 245–257, Dec 2000.

[4] D. Burger and T. Austin. The Simplescalar Toolset, Version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison, June 1997.

[5] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous Subordinate Microthreading (SSMT). In *Proceedings of ISCA*, 1999.

[6] T. Chen and J. Baer. Effective Hardware Based Data Prefetching for High Performance Processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[7] J.-L. Cruz, A. Gonzalez, M. Valero, and N. P. Topham. Multiple-Banked Register File Architectures. In *Proceedings of the 27th ISCA*, pages 316–325, 2000.

[8] D. Bailey, et al. The NAS Parallel Benchmarks. Technical Report TR RNR-94-007, NASA Ames Research Center, March 1994.

[9] M. Dubois and Y. H. Song. Assisted Execution. Technical Report CENG 98-25, EE-Systems, University of Southern California, Oct 1998.

[10] J. Dundas and T. Mudge. Improving Data Cache Performance by Pre-executing Instructions Under a Cache Miss. In *Proceedings of ICS*, 1997.

[11] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes. In *Proceedings of MICRO-31*, pages 59–68, 1998.

[12] K. Farkas, N. Jouppi, and P. Chow. Register File Considerations in Dynamically Scheduled Processors. In *Proceedings of HPCA*, 1996.

[13] N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of ISCA*, 1990.

[14] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2), March/April 1999.

[15] C.-K. Luk. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. In *Proceedings of the 28th ISCA*, 2001.

[16] C.-K. Luk and T. Mowry. Compiler-based Prefetching for Recursive Data Structures. In *Proceedings of ASPLOS VII*, pages 222–233, 1996.

[17] T. Monreal, A. Gonzalez, M. Valero, J. Gonzalez, and V. Vinals. Delaying Physical Register Allocation through Virtual-Physical Registers. In *Proceedings of MICRO-32*, pages 186–192, Nov 1999.

[18] M. Moudgill, K. Pingali, and S. Vassiliadis. Register Renaming and Dynamic Speculation: an Alternative Approach. In *Proceedings of MICRO*, 1993.

[19] T. Mowry, M. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of ASPLOS-V*, pages 62–73, 1992.

[20] V. Pai and S. Adve. Code Transformations to Improve Memory Parallelism. In *Proceedings of MICRO-32*, pages 147–155, 1999.

[21] S. Palacharla, N. Jouppi, and J. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of ISCA*, pages 206–218, 1997.

[22] S. Reinhardt and S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proceedings of the 27th ISCA*, pages 25–36, 2000.

[23] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting Dynamic Data Structures on Distributed Memory Machines. *ACM Transactions on Programming Languages and Systems*, Mar 1995.

[24] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proceedings of FTCS*, 1999.

[25] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace Processors. In *Proceedings of MICRO-30*, 1997.

[26] A. Roth, A. Moshovos, and G. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proceedings of ASPLOS VIII*, pages 115–126, 1998.

[27] A. Roth, A. Moshovos, and G. Sohi. Improving Virtual Function Call Target Prediction via Dependence-based Pre-computation. In *Proceedings of ICS*, 1999.

[28] A. Roth and G. Sohi. Speculative Data-Driven Multithreading. In *Proceedings of HPCA-7*, 2001.

[29] A. Sodani and G. Sohi. Dynamic Instruction Reuse. In *Proceedings of ISCA*, pages 194–205, 1997.

[30] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. In *Proceedings of ISCA*, 1995.

[31] J. Steffan and T. Mowry. The Potential for Using Thread Level Data-Speculation to Facilitate Automatic Parallelization. In *Proceedings of HPCA 4*, 1998.

[32] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Proceedings of ASPLOS-IX*, 2000.

[33] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of ISCA*, pages 392–403, 1995.

[34] S. Wallace and N. Bagherzadeh. A Scalable Register File Architecture for Dynamically Scheduled Processors. In *Proceedings of PACT*, Oct 1996.

[35] K. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2), April 1996.

[36] C. Zilles and G. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *Proceedings of ISCA*, pages 172–181, 2000.