An Instruction Set and Microarchitecture for Instruction Level Distributed Processing

Ho-Seop Kim and James E. Smith
Department of Electrical and Computer Engineering
University of Wisconsin—Madison
{hskim,jes}@ece.wisc.edu

Abstract

An instruction set architecture (ISA) suitable for future microprocessor design constraints is proposed. The ISA has hierarchical register files with a small number of accumulators at the top. The instruction stream is divided into chains of dependent instructions (strands) where intra-strand dependences are passed through the accumulator. The general-purpose register file is used for communication between strands and for holding global values that have many consumers.

A microarchitecture to support the proposed ISA is proposed and evaluated. The microarchitecture consists of multiple, distributed processing elements. Each PE contains an instruction issue FIFO, a local register (accumulator) and local copy of register file. The overall simplicity, hierarchical value communication, and distributed implementation will provide a very high clock speed and a relatively short pipeline while maintaining a form of superscalar out-of-order execution.

Detailed timing simulations using translated program traces show the proposed microarchitecture is tolerant of global wire latencies. Ignoring the significant clock frequency advantages, a microarchitecture that supports a 4-wide fetch/decode pipeline, 8 serial PEs, and a two-cycle inter-PE communication latency performs as well as a conventional 4-way out-of-order superscalar processor.

1 Introduction

During the past two decades, researchers and processor developers have achieved significant performance gains by finding and exploiting instruction level parallelism (ILP). Today, however, trends in technology, pipeline design principles, and applications all point toward architectures that rely less on increasing ILP and more on simpler, modular designs with distributed processing at the instruction level, i.e. *instruction level distributed processing* (ILDP) [25]. Technology trends point to increasing emphasis on on-chip interconnects and better power efficiency. Microarchitects are pushing toward pipelined designs with fewer logic levels per clock cycle (exacerbating interconnect delay problems and reducing sizes of single-cycle RAM) [1, 15]. Finally, design complexity has become a critical issue. In the RISC heyday of the mid-80s, the objective was a new processor design every two years; now it takes from four to six.

To study the full potential of future ILDP architectures, we are considering new instruction sets that are suitable for highly distributed microarchitectures. The goal is to avoid the encumbrances of instruction sets designed in a different era and with different constraints. This will enable us to study, in their simplest form, microarchitectures that are highly tolerant of interconnect delays, use a relatively small number of fast (high power) transistors, and support both very high clock frequencies and short pipelines. The resulting fast, lightweight processors will be ideal for chip multiprocessors supporting high throughput server applications [3] and for general-purpose processor cores that can drive highly integrated systems supporting various consumer applications.

The overall microarchitecture we propose consists of pipelined instruction fetch, decode, and rename stages of modest width that feed a number of distributed processing elements, each of which performs sequential in-order instruction processing. The instruction set exposes instruction dependences and local value communication patterns to the microarchitecture, which uses this information to steer chains of dependent instructions (*strands*) to the sequential processing elements. Dependent instructions executed within the same processing element have minimum communication delay as the results of one instruction are passed to the next through an accumulator. Taken collectively, the multiple sequential processing elements implement multi-issue out-of-order execution.

For applications where binary compatibility may not be a major issue (e.g. in some embedded systems), a new instruction set may be used directly in its native form. However, for general-purpose applications, a requirement of binary compatibility is a practical reality that must be dealt with. For general purpose applications there are two possibilities, both involve binary translation. One method is to perform on-the-fly hardware translation similar to the methods used today by Intel and AMD when they convert x86 binaries to micro-operations. Such a translation re-

1063-6897/02 $17.00 © 2002 IEEE

quires somewhat higher-level analysis than simple instruction mapping, however. Hence, the second method relies on virtual machine software, co-designed with the hardware and hidden from conventional software. This software can map existing binaries to the new ILDP instruction set in a manner similar in concept to the method used by the Transmeta Crusoe processor [19] and the IBM DAISY project [10]. A very important difference is that here the binary translation does not require the complex optimizations and scheduling that are used for VLIW implementations. Rather, the hardware we propose will be capable of dynamic instruction scheduling, so translation will involve a straightforward mapping of instructions. Consequently, the emphasis during translation is on identifying instruction inter-dependences and on making register assignments that reduce intra-processor communication. Binary translation can be performed either by a special co-processor [6] or by the main processor, itself.

2 Instruction Set and Microarchitecture

We begin with a brief description of the ILDP instruction set we propose. Following that is a description of the proposed microarchitecture. Then we discuss the specific features of both.

2.1 Instruction set overview

The proposed ISA uses a hierarchical register file. It has 64 general-purpose registers (GPRs) and 8 accumulators. We refer to the GPRs as registers R0-R63, and the accumulators as A0-A7. We focus here on the integer instruction set. Floating point instructions would likely use additional floating point accumulators, but would share the GPRs.

In any instruction, a GPR is either the source or destination, but not both; this is intended to simplify the renaming process. In addition, when an accumulator is used for the last time, i.e. becomes dead, this is specified in the instruction's opcode (possibly via a reserved opcode bit).

The instruction formats are given in Fig. 1. Instructions may be either 16 bits (one parcel) or 32 bits (two parcels).

2.1.1 Load/Store instructions

The memory access instructions load or store an accumulator value from/to memory. These instructions may only specify one GPR and one accumulator (as with all instructions). All load/store instructions are one parcel and do not perform an effective address addition.

```
Ai <- mem(Ai)
Ai <- mem(Rj)
mem(Ai) <- Rj
mem(Rj) <- Ai
```

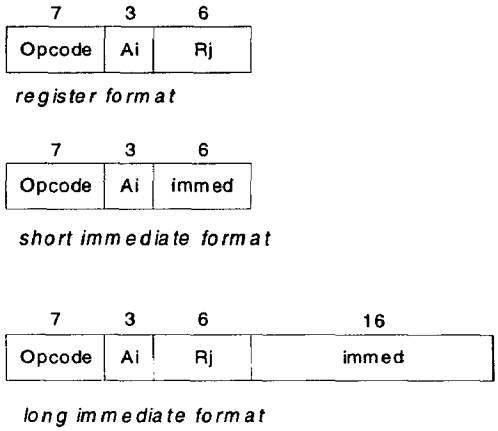


Figure 1. Instruction formats. Instructions are either 1 parcel (2 bytes) or 2 parcels (4 bytes).

2.1.2 Register instructions

The register instructions typically perform an operation on the accumulator and either a GPR or an immediate value, and the result is placed back in the accumulator. However, some instructions place the result into a GPR. Typical register instructions follow.

```
Ai <- Ai op Rj
Ai <- Ai op immed
Ai <- Rj op immed
Rj <- Ai
Rj <- Ai op immed
```

2.1.3 Branch/Jump instructions

The conditional branch instructions compare the accumulator with zero or the contents of a GPR. All the usual predicates (>, <, >=, <=, ==, !=) can be used. Branch targets are program counter (P) relative. The indirect jump is through either the accumulator or a GPR. For jump and link, the return address is always stored to a GPR.

```
P <- P + immed; Ai pred Rj
P <- P + immed; Ai pred 0
P <- Ai
P <- Rj
P <- Ai; Rj <- P++
```

2.1.4 Example

Figure 2 is a sequence of code from SPEC benchmark *164.gzip*, as compiled for the Alpha ISA. It is one of the more frequently executed parts of the program. Some Alpha instructions map to multiple ILDP instructions. However, the total number of bytes for instructions is slightly reduced. The Alpha version requires 40 bytes, and the ILDP version requires 36 bytes.

```
if (n) do {
        c = crc_32_tab[((int)c ^ (*s++)) & 0xff] ^ (c >> 8);
} while (--n);
```

a) C source code

Alpha assembly code	Equivalent register transfer notation	ILDP code
L1: ldbu t2, 0(a0)	L1: R2 <- mem(R0)	L1: A0 <- mem(R0)
subl a1, 1, a1	R1 <- R1 - 1	A1 <- R1 - 1
		R1 <- A1
lda a0, 1(a0)	R0 <- R0 + 1	A2 <- R0 + 1
		R0 <- A2
xor t0, t2, t2	R2 <- R2 xor R8	A0 <- A0 xor R8
srl t0, 8, t0	R8 <- R8 << 8	A3 <- R8 << 8
		R8 <- A3
and t2, 0xff, t2	R2 <- R2 and 0xff	A0 <- A0 and 0xff
s8addq t2, v0, t2	R2 <- 8*R2 + R9	A0 <- 8*A0 + R9
ldq t2, 0(t2)	R2 <- mem(R2)	A0 <- mem(A0)
xor t2, t0, t0	R8 <- R2 xor R8	A0 <- A0 xor R8
		R8 <- A0
bne a1, L1	P <- L1, if (R1 != 0)	P <- L1, if (A1 != 0)

b) Alpha assembly code, equivalent register transfer notation, and corresponding ILDP code

Figure 2. Example program segment from benchmark *164.gzip*

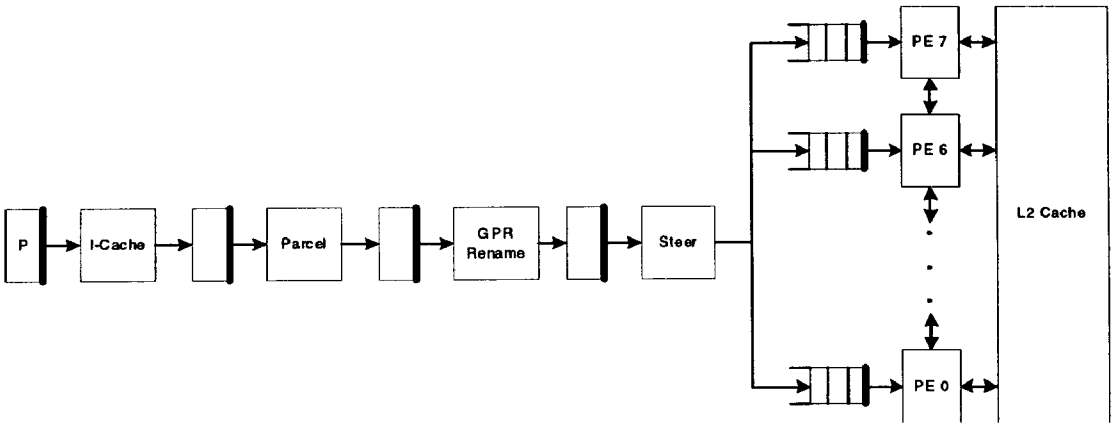
2.2 Microarchitecture

The concept behind the instruction set is that the dynamic program dependence graph can be decomposed into strands – chains of dependent instructions. Instructions in each strand are linked via an accumulator. The strands communicate with each other through the GPRs.

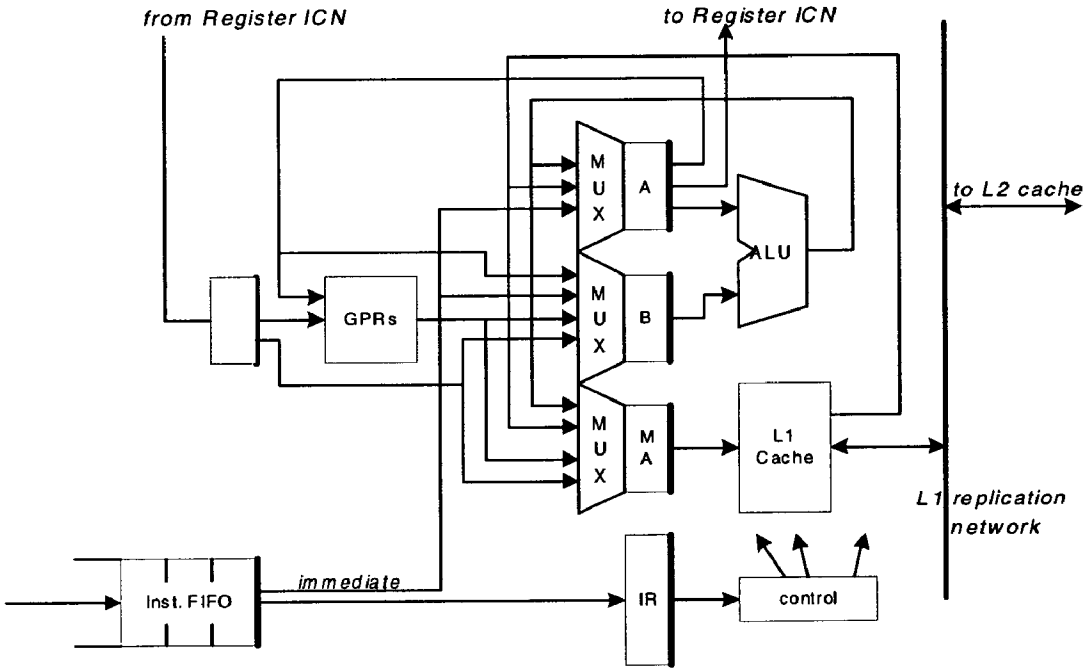
The strand per accumulator concept is reflected in the microarchitecture. Referring to Fig. 3a, instructions are fetched, parceled, renamed, and steered to one of eight processing elements. Instructions will be *fetched* four words (16 bytes) at a time. However, in most cases these four words contain more than 4 instructions. After I-fetch, the remainder of the instruction decode/rename pipeline is four instructions wide. *Parceling* is the process of identifying instruction boundaries and breaking instruction words into individual instructions. One simplification we are considering is to have instructions start and end on a cache line (at least 8 words) boundary. This will avoid instruction words spanning cache line (and page) boundaries – an unnecessary complication. The *renaming* stage renames only GPRs. The accumulators are not renamed at this stage; they undergo a simpler type of renaming as a byproduct of steering to the sequential processing elements.

The *steering* logic directs strands of renamed instructions to one of eight issue FIFOs, depending on the accumulator to be used. Each FIFO feeds a sequential processing element with its own internal physical accumulator (Fig. 3b). Any instruction that has an accumulator as an output, but not as an input, is steered to the first empty FIFO; consequently, the logical accumulator is renamed to the physical accumulator. Any later instruction that uses the same accumulator as an input is steered to the same FIFO. Whenever all the FIFOs are non-empty and a new accumulator is called for, the steering process either stalls, or uses a heuristic to choose a FIFO to use next. If a FIFO has a "dead" accumulator instruction at its tail, then instructions from a new strand can be steered into the FIFO. A good heuristic is to use the first such FIFO likely to go empty (e.g. the one with the fewest instructions).

The instructions in a FIFO form a dependence chain, and therefore will issue and execute sequentially. The instruction at the head of the FIFO passes through the GPR pipeline stage, reads its GPR value (if available), and moves into the issue register (IR). If the GPR value is not available, the instruction waits in the IR until the value becomes available. Hence, the IR is like a single reservation station with a single data value entry. When its GPR value is available, the instruction issues and begins execution. Fig. 3b shows an ALU and an L1 cache as functional units; in practice there will also be a shifter, and some other units. There is no contention/arbitration for any of the replicated functional units. A sequential control unit drives the processing element. Note that the functional units can be single or multi-cycle, but do not require pipelining. Because accumulator values stay within the same processing element, they can be bypassed without additional delay. However, GPR values produced in one PE must be communicated to the others. This will take additional clock cycles. The network for communicating GPR values can be a bus, a ring, or point-to-point. As will be shown, the bandwidth requirements are very modest and performance is relatively insensitive to this latency. The PE in the figure has two write ports to the GPR file; this will avoid contention between the accumu-



a) Block diagram of ILDP processor



b) Processing element (ICN stands for interconnection network)

Figure 3. The distributed processor and detail of processing element

lator path and the GPR interconnection network. However, an alternate design could arbitrate the two and have a single write port. Simulation results in Section 4.3 will explore GPR write bandwidth requirements and the importance of inter-PE latency for communicating GPR values.

We plan to replicate the (small) low-latency L1 data cache and use a replication network to keep the contents of all the L1 caches equal (within a 2 clock period window as values are communicated). The L1 cache is fed directly from the issue stage of the PE because the memory instructions do not perform effective address additions. Because the PEs are sequential, issue bandwidth within a PE is not a critical resource as it is in a conventional superscalar processor, so issuing two instructions (in two cycles) for those load/stores that require address additions does not pose a performance problem. However, it does provide a performance *advantage* for those loads where an effective address addition is not needed (statistics are given in Section 4). Having the memory address available at issue time has other advantages; for example, store address queue checking can be done as part of the issue function, in much the same way as the Cray-1 does memory bank conflict checking at issue time [7]. Block-

	ILDP microarchitecture	4-way out-of-order superscalar
Parcel stage	Yes (if 2 inst. Lengths are used)	No
Decode bandwidth	4 instructions per cycle	4 instructions per cycle
Rename bandwidth	Total 4 read or write ports to map table	Total 12 read or write ports to map table
Steering logic	Simple, based on accumulator number	Complex dependence-based heuristic (if clustered)
Issue logic	Sequential in-order issue	4-way out-of-order issue from 128-entry RUU
Register file	2 write ports, 1 read port	4 write ports, 8 read ports
Bypasses	N x 2; for N functional units in a PE	M x 8; for M total functional units (M > N)

Table 1. Complexity comparison: ILDP processor vs. conventional out-of-order superscalar processor

ing on an L1 cache miss can also be done by blocking issue of the next instruction.

Each PE has a copy of the store address queue for memory disambiguation (not shown in the figure). Every load instruction consults the queue for possible conflicts with preceding store instructions. If all previous store addresses are known and do not conflict with the load, the load instruction is allowed to issue. Store address queue entries are allocated prior to the steering stage. As store addresses are computed, the address bits (or, for simplicity, a subset of 8-16 bits) are communicated to the replicated store queues.

Both GPRs and accumulators need to be rolled back when there is an exception or a branch misprediction. A conventional reorder buffer-based recovery mechanism can be used for GPRs; the GPR rename map is restored to the exception/misprediction point. To recover accumulators, produced accumulator values are buffered by a FIFO inside a PE. In effect, this is a small history buffer. When an instruction retires, the previous older value of the instruction's accumulator is also retired from the accumulator value FIFO. Should there be an exception, the oldest entries in the accumulator value FIFOs (or architectural accumulators) are rolled back. Similar to the GPR rename map recovery, steering information is also rolled back from the reorder buffer. Recovery from a mispredicted branch is done similarly. Here accumulator steering information can be recovered either from branch checkpoints or by sequential roll back from the saved information in the reorder buffer entries.

2.3 Discussion

As stated earlier, we are targeting a microarchitecture that will be simple and provide very high performance through a combination of a very fast clock and modest ILP. Because the clock cycle of an aggressive design depends on the details of every pipeline stage, we prefer not to use a few gross aspects of the design (e.g. bypasses, issue logic) to verify a quantitative clock cycle estimate. We prefer to let the simplicity stand as self-evident and defer clock cycle estimates until we have done some detailed, gate-level design. Table 1 compares complexity of several of the key functions with a baseline superscalar processor (similar to the one we use in Section 4 for performance comparisons). The ILDP microarchitecture complexities are given on a per PE basis, because that is the complexity that will ultimately determine the clock cycle.

Variable length instructions are aimed at reducing the instruction footprint to permit better I-cache performance, especially if a small single-cycle L1 I-cache is used. Although it may seem un-RISC-like, variable length instructions were a hallmark of the original RISCs at Control Data and Cray Research. That a RISC should have single-length instructions has intuitive appeal, but to some extent it was a 1980s reaction to the very complex multi-length instruction decoding required by the VAX. There is little real evidence that having only one instruction size is significantly better than having a small number of easily decoded sizes. It is entirely possible that the advantages of a denser instruction encoding and more efficient I-fetch bandwidth may outweigh the disadvantage of having to parcel a simple variable-width instruction stream [12]. This is *not* a key part of our research, however, but we feel it is worth exploring, and if it appears that a single instruction size is the better approach, we can go back to all 32-bit instructions with relatively little effort.

3 Strand Formation

The most important aspect of strand formation is assignment of values to GPRs and accumulators, because, in effect, these assignments drive strand formation itself. To understand the basic idea of accumulator-oriented register assignment it is helpful to consider important register usage patterns, see Fig. 4.

We are interested in separating register values that have a relatively long lifetime and are used many times from those that are used only once or a small number of times in close proximity [13]. In general, the former values should be placed in GPRs and the latter in accumulators. A useful heuristic is to find register values that are consumed multiple times by the same static instruction. These are defined to be *static global* register values and will be assigned to GPRs (Ri in the figure). All the other register values will be considered *local* and will be assigned to accumulators, Ak and An in the figure. If a local value is used only once, then it will never be placed in a GPR. However, if one of these local values is used by

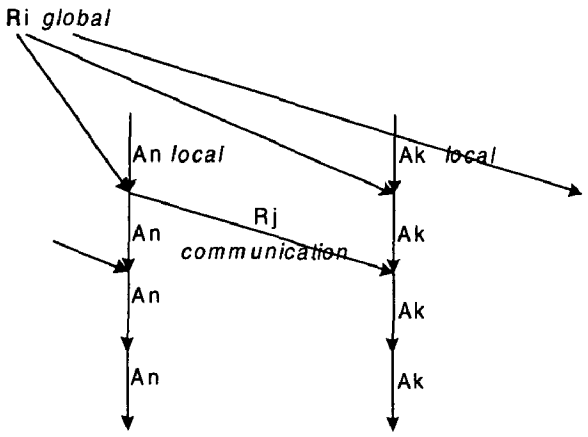


Figure 4. Types of values and associated registers. *Static global* and *communication global* values are held in GPRs, and *local* values are held in accumulators.

more than one consuming instruction, then a copy will be made to a GPR thereby communicating the value to other strands. These are referred to as *communication globals*.

If binary compatibility is not required, a static compiler can be used to generate programs in the native ILDP ISA. The compiler will allocate temporary register values to accumulators while keeping the rest in the GPRs. The compiler performs this non-conventional register allocation based on register usage and lifetime analysis.

In our research, dynamic binary translation is used to form strands. Currently we are using Alpha instruction binaries as the source ISA for translation. For our initial studies, we are analyzing simulator-generated instruction traces, but in the future we plan to use the basic method in [10, 16, 19], where interpretation is first used to produce profile data that is fed to a translator.

Given a code sequence and the above profile information that identifies static global values, strand formation is implemented via a single-pass linear scan of instructions to be translated; currently we do not perform code reordering during translation. For this discussion, we assume the source instruction set is RISC-like (Alpha in particular), with two source registers and one destination register, and simple addressing modes. At any given point in the scan, some register values are held in accumulators and others are held in GPRs.

When the scan reaches an instruction that has no input value currently assigned to an accumulator, it begins a new strand. If an instruction has a single input value assigned to an accumulator, then the same accumulator is used as the instruction's input, and the instruction is added to the producer's strand. If there are two input values assigned to accumulators, then two strands are intersecting. At this point, one of the strands is terminated by copying its accumulator into a GPR (to be communicated to the other strand). The other strand continues with the already-assigned accumulator and the just-assigned GPR

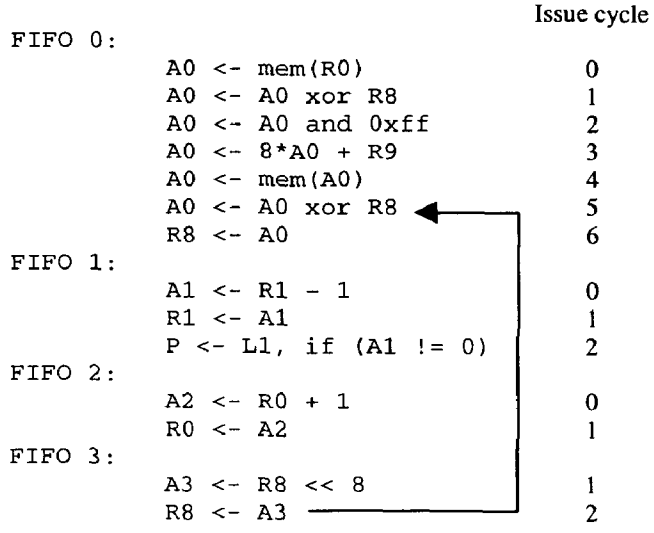


Figure 5. Issue timing of the example code

as inputs. To decide which strand to terminate, a good heuristic is to follow the strand that is longer, up to that point, to avoid introducing the communication latency into the already longer strand.

If an instruction has its output assigned to a static global register (or has no output value) the strand is terminated. If the new strand requires an accumulator when all the accumulators are live, then one of the live strands is terminated by copying its accumulator into a GPR. We choose the longest active strand as the victim. This tends to balance the lengths of the strands in the FIFOs.

Example

We complete this section with a continuation of the code example given in Section 2.1.4. Four accumulators are used (A0 through A3), so the instructions are steered to four different processing elements (FIFOs) as shown in Fig. 5. The strands are relatively independent, except where two strands converge to form inputs to the second *xor*. For this short code sequence, 14 instructions are issued in six clock cycles.

4 Evaluation

This section contains experimental evaluation of the instruction set properties and microarchitecture design decisions made in the previous sections.

4.1 Simulation methodology

To evaluate the proposed ILDP instruction set and microarchitecture, we first developed a functional simulator, which will be referred to as the *profiler* to distinguish it from the timing simulator. The profiler runs Alpha 21264 programs and profiles the dynamic register value usage. If a load/store instruction uses a non-zero immediate field

Bench-mark	Alpha instruc-tion count	% of in-structions w/ zero or one input register operand	% of loads w/o imme-diate	% of stores w/o im-mediate
164.gzip	3.50 bil.	55.09	43.6	50.6
175.vpr	1.54 bil.	51.16	34.9	29.1
176.gcc	1.89 bil.	62.38	34.8	15.8
181.mcf	260 mil.	57.74	30.4	11.9
186.crafty	4.18 bil.	54.34	27.0	13.4
197.parser	4.07 bil.	57.68	44.8	22.2
252.eon	95 mil.	55.83	15.7	15.4
254.gap	1.20 bil.	61.60	44.9	27.1
300.twolf	253 mil.	50.48	41.5	31.2

Table 2 Benchmark program properties

(i.e. requires an address add), then the instruction is split into two instructions for address calculation and memory access. Alpha conditional move instructions require three source operands and are also split into two instructions. Note that the Alpha 21264 processor similarly splits conditional moves into two microinstructions [18]. Static global registers are first identified using the heuristic of selecting all registers that are consumed multiple times by the same static instruction. Strands are then identified and accumulators and communication globals are assigned using the method described in the previous section.

Simulation tools were built on top of the SimpleScalar toolset 3.0B [4]. The profiler maps the strand-identified Alpha traces into ILDP traces and feeds the timing simulator. The timing simulator models the proposed microarchitecture and executes translated traces of ILDP instructions.

We selected nine programs from the SPEC 2000 integer benchmarks compiled at the base optimization level (`-arch ev6 -non_shared -fast`). The compiler flags are same as those reported for Compaq AlphaServer ES40 SPEC 2000 benchmark results. DEC C++ V.6.1-027 (for *252.eon*) and C V.5.9-005 (for the rest) compilers were used on Digital UNIX 4.0-1229. The `test` input set was used and all programs were run to completion.

4.2 Strand Characteristics

Table 2 contains some general statistics from the Alpha traces. More than half of the dynamic instructions have zero or one input register operand. Loads, conditional branches, and integer instructions with an immediate operand belong to this category. This statistic implies the data dependence graphs are rather "thin", with relatively little inter-strand communication. We also note there are substantial numbers of load and store instructions that do not require address calculation. With the proposed ILDP instruction set, these instructions can bypass the address addition and be sent to the data cache directly.

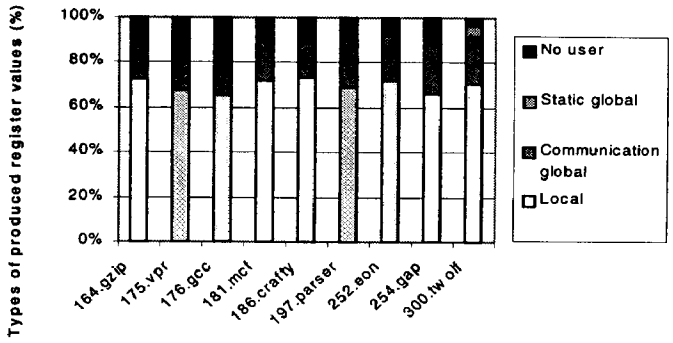


Figure 6. Types of register values

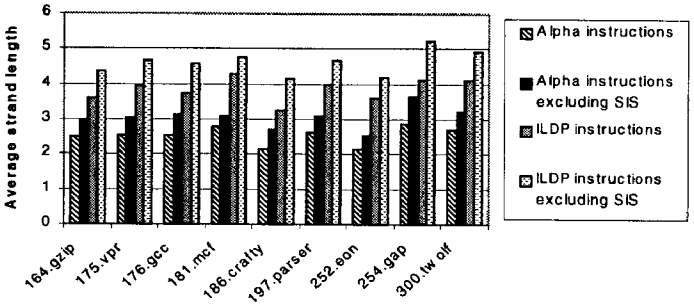


Figure 7. Average strand lengths

We also collected data regarding types of register values: static global, communication global, and local. Fig. 6 shows the fraction of values for each of the three classes (and those that have no consumers). Most values produced are locals (about 70%). Since these values are used only once, they do not have to leave their processing element and do not consume GPR bandwidth. Only about 20% of the values have to be placed in global registers (which suggests relatively low global register write bandwidth). Finally, 10% of produced values are never used. Some of these come from high-level program semantics; for example, a function's return value or some of its input arguments might not be used, depending on the program control flow. Also aggressive compiler optimizations, e.g. hoisting instructions above branches, sometimes result in unused values.

Fig. 7 shows the lengths of strands measured in both Alpha and ILDP instructions. Average strand size is 3.85 in ILDP instructions or 2.54 Alpha instructions. There are many single-instruction strands (SIS in the figure) that do not contribute much to the total number of instructions but affect the average strand size significantly. These single-instruction strands include unconditional branch and jump instructions, and instructions whose produced value is not used. An important characteristic of the single-instruction strands is that they can be steered to any FIFO; no other instruction depends on them. If the single-instruction strands are ignored, the average size of strands is 4.62 in ILDP instructions and 3.04 in Alpha instructions.

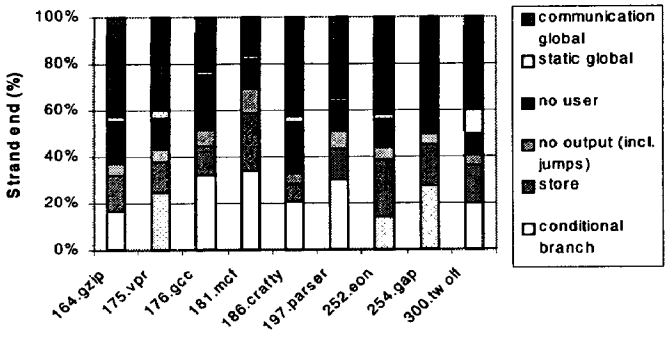


Figure 8. Strand end

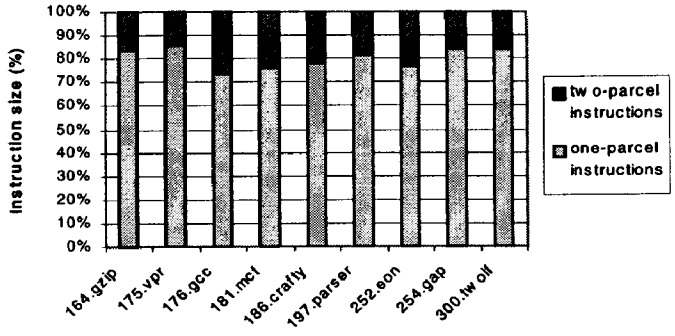


Figure 9. Instruction size

It is also interesting to see how strands end (Figure 8). About 35 to 70% of the strands have a "natural" ending – dependence chains leading to conditional branch resolution and store address/value calculation. About 20 to 45% produce communicated globals.

Finally we look at the impact of using two instruction sizes in translated ILDP ISA traces. Fig. 9 shows that on a dynamic basis, single parcel instructions account for 73 to 85% of total ILDP instructions, resulting in an average of 2.39 bytes per instruction.

4.3 Performance Results

Timing simulations are trace-driven. Because of different instruction sizes, it was easiest to simply assume an ideal I-cache for both the baseline superscalar and the ILDP microarchitecture. (All nine Alpha benchmark programs with the `test` input set have an L1 I-cache miss rate less than 0.7% with a 2-way set-associative, 32KB, 64-byte line size I-cache; less than 3.8% for an 8KB I-cache). We also believe the performance results from the trace-driven simulations will closely track those of a true dynamically translated system because program execution is dominated by small number of repeating strands in most cases. Fig. 10 shows more than 95% of total executed instructions belong to the strands that repeat more than 1000 times.

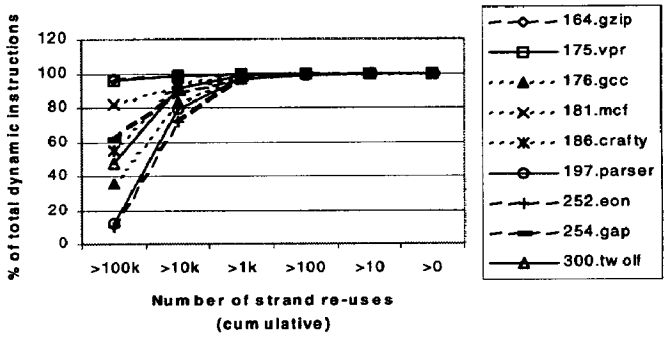


Figure 10. Cumulative strand re-use

Simulator configurations are summarized in Table 3.

	ILDP microarchitecture	Out-of-order superscalar processor
Branch prediction	16K entry, 12-bit global history Gshare predictor 3-cycle fetch redirection latencies for both misfetch and misprediction	
I-cache, I-TLB	Ideal	
D-cache, D-TLB	L1: 2-way set-assoc., 8KB size, 64-byte line size, 1-cycle latency, random replacement	L1: 4-way set-assoc., 32KB size, 64-byte line size, 2-cycle latency, random replacement
	L2: 2-way set-assoc., 256KB size, 128-byte lines, 8-cycle latency, random replacement TLB: 4-way set-associative, 32-entry, 4KB page size, LRU replacement	
Memory	72-cycle latency, 64-bit wide, 4-cycle burst	
Reorder buffer size	128 ILDP instructions	128 Alpha instructions
Fetch/decode/retire bandwidth	4 ILDP instructions	4 Alpha instructions
Issue window size	8 (FIFO heads)	128 (same as the ROB)
Issue bandwidth	8	4 or 8
Execution resources	8 fully symmetric functional units	4 or 8 fully symmetric functional units
Misc.	2 or 0 cycle global communication latency	No communication latency, oldest-first issue

Table 3. Simulator configurations

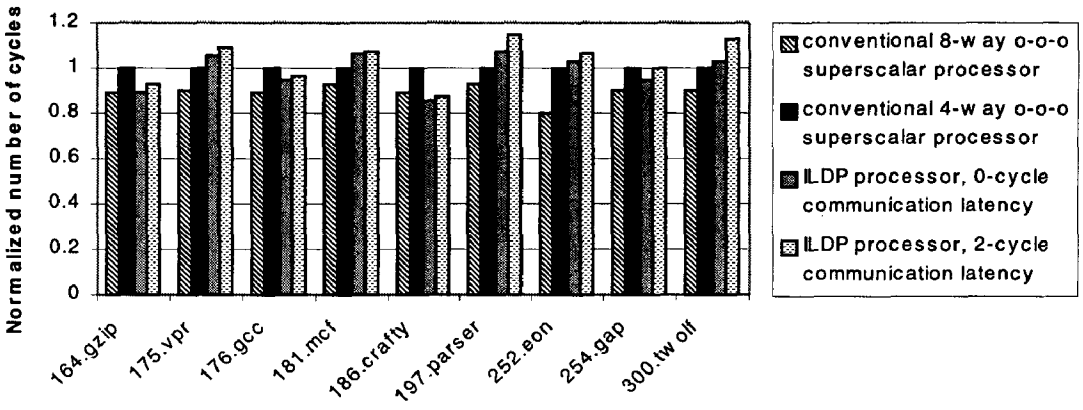


Figure 11. Normalized number of cycles with the 4-way with superscalar processor as the baseline

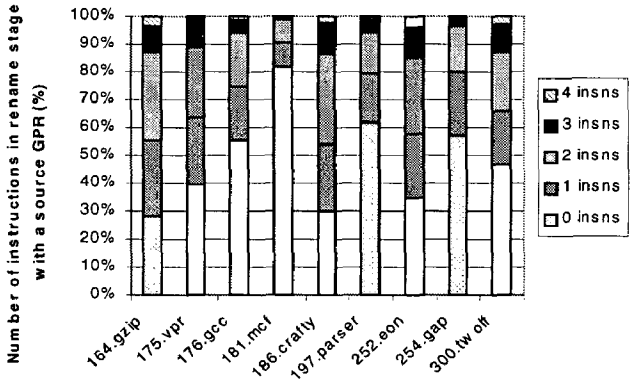


Figure 12. Global register rename map read bandwidth

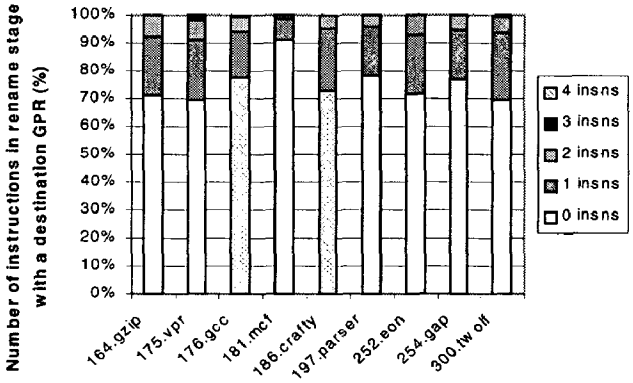


Figure 13. Global register rename map write bandwidth

Note that in keeping with the philosophy of smaller/faster memory structures, the L1 caches in the proposed microarchitecture are one quarter the size of the superscalar counterpart. The latency difference of one cycle results from the ILDP microarchitecture not having an address addition in the load path (as described previously); for those loads that require an add, the latencies are effectively the same.

Because the two microarchitectures being simulated have different ISAs, Instructions Per Cycle (IPC) is not a good metric for comparing performance. Instead the total number of cycles is used.

The results show that the proposed microarchitecture performs approximately as well as a conventional 4-way out-of-order superscalar processor *ignoring any clock frequency advantage*, which, based on Table 1 and the smaller data cache should be considerable.

More importantly, the proposed microarchitecture is tolerant of global communication latency. There is only a 2 to 7 percent performance degradation as communication latency increases from 0-cycles to 2-cycles. In most cases global communication latency is not imposed on the critical path by the dependence-based strand formation algorithm.

For *164.gzip* and *186.crafty*, the proposed microarchitecture with 0-cycle communication latency outperforms an 8-way out-of-order superscalar processor despite the reduced issue freedom of the proposed microarchitecture. This comes primarily from the reduced load latency for as many as 43.6% of loads where there is no need for an effective address addition.

To further understand implementation complexity issues, we collected statistics related to rename, steering, and global register bandwidths. Fig. 12, 13 show the global register rename bandwidths per cycle. With a four-wide pipeline, over 95% of the time three or fewer global register mappings are read, and over 90% of the time only zero or one register mapping is updated. This suggests that three read ports and one write port in the mapping table will likely be sufficient *if* we want to add the complexity of adding stall logic.

A significant complexity issue is the number of write ports to the GPR file. Using accumulators for local values greatly reduces the required GPR bandwidth. Collected GPR write bandwidth statistic closely follows Fig. 13; it shows one register write port is enough more than 95% of time. Hence, if we are willing to add arbitration for a single write port, then the GPR can be reduced to a single

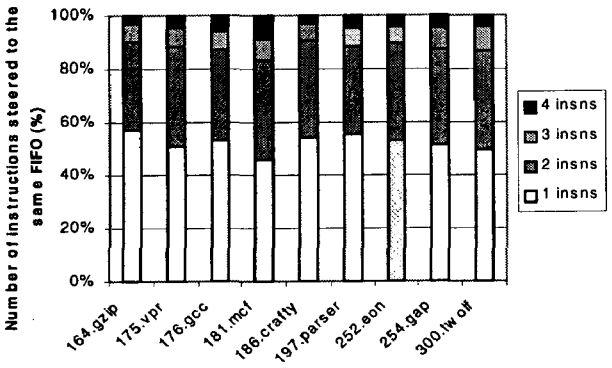


Figure 14. Number of instructions steered to the same FIFO at the same cycle

read port and a single write port with little performance impact.

Although the steering logic is simplified by only considering accumulator names in making steering decisions, the number of instructions steered to any one FIFO can also affect complexity. We measured the number of ILDP instructions steered to the same FIFO during the same cycle. The results in Fig. 14 show that two or fewer instructions are steered to the same FIFO over 82% of the time.

5 Related Work

The instruction set is very much inspired by the S. Cray scalar ISAs (just as the 1980s microprocessor RISCs were). However, in a sense, we follow the Cray ISAs more closely than the microprocessor-based RISCs. In particular, we use hierarchical register files with a very small file at the top of the hierarchy, variable length instructions, and in-order instruction issue (albeit within individual processing elements). Even though the technology was quite different when Cray's designs were undertaken, the issues of interconnect delays, power consumption, and design complexity were of critical importance, just as they are today, and will be in the future. In effect, the proposed ILDP ISA is a cross product of two Cray-2 designs. One is an abandoned Cray-2 design [8] that had a single re-named accumulator and a general register file of 512 elements. The completed Cray-2 design [9] had 8 integer registers, 64 lower level registers, and used conventional 3 operand instructions.

The ZS-1 [24] was an early superscalar design with instructions issuing simultaneously from two FIFOs, motivated by issue logic simplicity. The RS/6000 [2] used a similar design. In [22] a dependence-based microarchitecture that issues instructions from multiple FIFOs was proposed. That work, and others [5, 11] proposed clustered microarchitectures to localize register communication. Trace processors [23, 27] are another form of distributed microarchitecture, with each processing element being a simple superscalar processor. Trace processors also support a hierarchy of register files for local and global communication. Similarly, the multiscalar paradigm [14, 26] was designed with a goal of processor scalability and used a number of innovative distributed processing features. The PEWS mechanism [17] also uses dependence-based instruction steering but uses versioning for both registers and memory.

RAW architecture [20] and Grid Processor Architecture [21] propose network-connected tiles of distributed processing elements running programs compiled for new ISAs that expose underlying parallel hardware organization. Both architectures are targeted to achieve high ILP on scalable hardware. As such, both are sensitive to communication latency and depend heavily on the compiler. In contrast, our aim is to achieve high performance in general purpose applications with a combination of a very high clock frequency, moderate ILP and a relatively conventional compiler target.

IBM DAISY [10] and Transmeta Crusoe [19] use dynamic binary translation to run legacy software on the hardware that executes different instruction set. Ebcioglu et al. [10] showed the translation overhead is negligible. Both use VLIW as underlying implementation; as a result, the run-time software performs extensive instruction rescheduling to achieve desirable ILP on in-order VLIW implementation.

6 Conclusions and Future Research

For future processors, we propose an instruction set that exposes inter-instruction *communication* and is targeted at a distributed microarchitecture with both short pipelines and high frequency clocks. A primary goal is high performance by using a small number of logic transistors. This is counter to the conventional trend that uses instruction sets that expose instruction *independence*; use very long (deep) pipelines, and high logic transistor counts. The major challenge is not to think of enhancements that consume transistors and yield small incremental performance gains, but to develop an overall paradigm that achieves high performance through simplicity.

The overall microarchitecture we propose consists of a number of distributed processing elements, each of which is a simple in-order pipeline. By using an accumulator-based instruction set, the hardware implementation can steer chains of dependent instructions, "strands", to the simple in-order issue processing elements. In aggregate, the multiple in-order processing elements enable superscalar out-of-order execution as each of the processing elements adapts to the delays it encounters.

In this paper we have demonstrated that a distributed microarchitecture is capable of IPC performance levels that are roughly equivalent to a homogeneous 4-to-8-way superscalar processor. Most of the processing stages – renaming, register access, issuing, bypassing, data cache –

are much simpler than in a conventional superscalar processor, however, and the prospects for a much faster clock frequency are very good.

The distributed microarchitecture has other advantages that have not been discussed thus far. First, it will be amenable to multiple clock domains, which may be asynchronous with respect to one another. This is a very important feature for an aggressively clocked implementation where clock skew is critical. Second, it is also amenable to microarchitecture-level clock- and power-gating. Some of the processing elements can be gated off when not needed to save power.

In the future we plan to explore the proposed ISA and microarchitecture greater detail. First, we plan to perform a gate level design of the integer proportion of the proposed processor. This will validate claims of simplicity. Second, we plan to implement a binary translation infrastructure to allow on-the-fly translation of existing program binaries. Then, we will be able to provide accurate overall performance estimates that will demonstrate the feasibility of this overall approach.

7 Acknowledgements

We would like to thank Timothy H. Heil and Martin J. Licht for discussions and S. Subramanya Sastry for help on the initial version of the profiling simulator. This work is being supported by SRC grants 2000-HJ-782 and 2001-HJ-902, NSF grants EIA-0071924 and CCR-9900610, Intel and IBM.

8 References

[1] V. Agrawal et al., "Clock Rate vs. IPC: The End of the Road for Conventional Microarchitectures", *27th Int. Symp. on Computer Architecture*, pp. 248-259, Jun 2000.

[2] H. Bakoglu et al., "The IBM RISC System/6000 Processor: Hardware Overview", *IBM Journal of Research and Development*, pp. 12-23, Jan 1990.

[3] L. Barroso et al., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing", 27th Int. Symp. on Computer Architecture, pp. 282-293, Jun 2000.

[4] D. Burger et al., "Evaluating Future Microprocessors: The SimpleScalar Toolset", *Tech. Report CS-TR-96-1308*, Univ. of Wisconsin—Madison, 1996.

[5] R. Canal et al., "A Cost-Effective Clustered Architecture", *Int. Conf. On Parallel Architectures and Compilation Techniques (PACT99)*, pp. 160-168, Oct 1999.

[6] Yuan Chou and J. Shen, "Instruction Path Coprocessors", *27th Int. Symp. on Computer Architecture*, pp. 270-279, Jun 2000.

[7] CRAY-1 S Series, Hardware Reference Manual, Cray Research, Inc., *Publication HR-808*, Chippewa Falls, WI, 1980.

[8] CRAY-2 Central Processor, *unpublished document*, circa 1979. http://www.ece.wisc.edu/~jes/papers/cray2a.pdf.

[9] CRAY-2 Hardware Reference Manual, Cray Research, Inc., *Publication HR-2000*, Mendota Heights, MN, 1985.

[10] K. Ebcioglu et al., "Dynamic Binary Translation and Optimization", *IEEE Trans. on Computers*, Vol. 50, No. 6, pp. 529-548, Jun 2001.

[11] K. Farkas et al., "The Multicluster Architecture: Reducing Cycle Time Through Partitioning", *30th Int. Symp. on Microarchitecture*, pp. 149-159, Dec 1997.

[12] M. Flynn, *Computer Architecture: Pipelined and Parallel Processor Design*, Jones and Bartlett Publishers, pp. 109-132, 1995.

[13] M. Franklin and G. Sohi, "Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors", *25th Int. Symp. on Microarchitecture*, pp. 236-245, Dec 1992.

[14] M. Franklin and G. Sohi, "The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism", *19th Int. Symp. on Computer Architecture*, pp. 58-67, Dec 1992.

[15] R. Ho et al., "The Future of Wires", *Proceedings of the IEEE*, Vol. 89, No. 4, pp. 490-504, Apr 2001.

[16] R. Hookway and M. Herdeg, "DIGITAL FX!32: Combining Emulation and Binary Translation", *Digital Technical Journal*, Vol. 9, No. 1, pp. 3-12, 1997.

[17] G. Kemp and M. Franklin, "PEWs: A Decentralized Dynamic Scheduler for ILP Processing", *Int. Conf. On Parallel Processing*, pp. 239-246, Aug 1996.

[18] R. E. Kessler, "The Alpha 21264 Microprocessor", *IEEE Micro*, Vol. 19, No. 2, pp. 24-36, Mar/Apr 1999.

[19] A. Klaiber, "The Technology Behind Crusoe Processors," *Transmeta Technical Brief*, 2000.

[20] W. Lee et al., "Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine", *8th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 46-57, Oct 1998.

[21] R. Nagarajan et al., "A Design Space Evaluation of Grid Processor Architectures", *34th Int. Symp. on Microarchitecture*, pp. 40-51, Dec 2001.

[22] S. Palacharla et al., "Complexity-Effective Superscalar Processors," *24th Int. Symp. on Computer Architecture*, pp. 206-218, Jun 1997.

[23] E. Rotenberg, et al., "Trace Processors", *30th Int. Symp. on Microarchitecture*, pp. 138-148, Dec 1997.

[24] J. E. Smith et al., "Astronautics ZS-1 Processor", *2nd Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 199-204, Oct 1987.

[25] J. E. Smith, "Instruction-Level Distributed Processing", *IEEE Computer*, Vol. 34, No. 4, pp. 59-65, Apr 2001.

[26] G. Sohi et al., "Multiscalar Processors", *22nd Int. Symp. on Computer Architecture*, pp. 414-415, Jun 1995.

[27] S. Vajapeyam and T. Mitra, "Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences", *24th Int. Symp. on Computer Architecture*, pp. 1-12, Jun 1997.