A Technology-Scalable Architecture for Fast Clocks and High ILP

Karthikeyan Sankaralingam   Ramadass Nagarajan   Stephen W. Keckler   Doug Burger
Computer Architecture and Technology Laboratory
Department of Computer Sciences
Tech Report TR2001-02
The University of Texas at Austin
cart@cs.utexas.edu — www.cs.utexas.edu/users/cart

ABSTRACT

CMOS technology scaling poses challenges in designing dynamically scheduled cores that can sustain both high instruction-level parallelism and aggressive clock frequencies. In this paper, we present a new architecture that maps compiler-scheduled blocks onto a two-dimensional grid of ALUs. For the mapped window of execution, instructions execute in a dataflow-like manner, with each ALU forwarding its result along short wires to the consumers of the result. We describe our studies of program behavior and a preliminary evaluation that show that this architecture has the potential for both high clock speeds and high ILP, and may offer the best of both the VLIW and dynamic superscalar architectures.

1 Introduction

Conventional microarchitectures have been improving in performance by approximately 50-60% per year, improving the instructions per cycle (IPC) using more transistors on a chip and increasing the clock speed. However both strategies will fail for future technologies (50nm and below), with clock speed growth slowing down because of fundamental pipelining limits and wire delays making architectures communication bound [1]. Thus, today's architectures will not scale, showing diminishing returns in IPC even with increasing chip transistor budgets. New designs must address these issues, efficiently utilizing the increasing transistor budget while overcoming communication bottlenecks.

One approach for extracting ILP is through conventional superscalar cores that detect parallelism at run-time. The amount of ILP that can be detected is limited by the issue window, whose logic complexity grows as the square of the number of entries [12]. Conventional architectures also rely on many frequently accessed global structures, such as register files, re-order buffers and issue windows, which become bottlenecks limiting clock speed or pipeline depths.

Another approach for extracting parallelism is taken by VLIW machines, in which ILP analysis is performed at compile time. Instruction scheduling is performed by the compiler, orchestrating the flow of execution statically. This approach performs well only for regular workloads and suffers from the drawback that dynamic events are not handled well — a stall in one functional unit forces the entire machine to stall, since all functional units must be synchronized.

In this paper, we describe a new architecture called the Grid Processor that takes into consideration the technology constraints of wire delays and pipelining limits. The compiler is used to detect parallelism and statically schedule instructions on a computation substrate, but instructions are issued dynamically. We propose an execution substrate that consists of a set of *named distributed computing elements* to which the compiler statically assigns individual instructions.

The architecture does not suffer from VLIW issue restrictions, as instructions are issued dynamically and executed in a dataflow fashion. Instructions from a compiler-generated basic block or hyperblock are mapped statically to nodes in the computation array, with each node being assigned one or more instructions. The nodes issue instructions dynamically when the input operands are available. Temporary values produced and consumed inside a block are not visible to the architectural state, and are instead forwarded directly from the producers to their consumers.

We propose a fine-grained partitioning of the issue window and associated functional units (FU). The computation array includes both a grid of issue window-FU pairs (nodes) and a dedicated communication network for passing data. Data produced at a node are routed dynamically through intermediate nodes to their eventual destinations. The architecture is a hybrid between conventional superscalar and conventional VLIW architectures, issuing instructions dynamically with static scheduling. The hardware substrate is designed to extract high ILP and run at aggressive clock rates.

In the Grid Processor, the available transistor budget is used to build an array of computation elements aimed at overcoming several challenges of communication overhead in future systems. First, by forwarding values directly between producers and consumers, the reliance on centralized structures is reduced. Second, compiler controlled physical layout ensures that the critical path is scheduled along the shortest physical path. Finally, instruction blocks are mapped onto the grid as single units of computation amortizing scheduling and decode overhead over a large number of instructions. The reduced reliance on centralized structures allows the computation substrate to be clocked at high speeds.

The remainder of this paper is organized as follows. Section 2 describes the key features of the

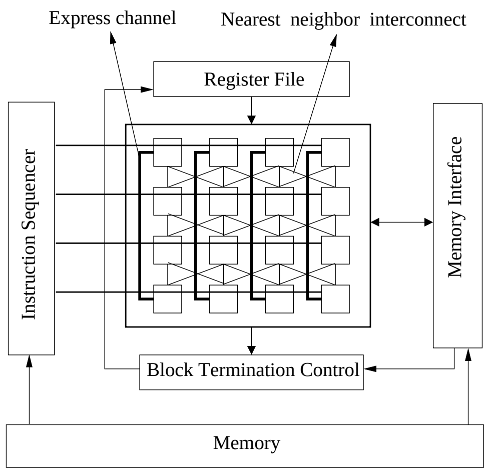


Figure 1: Grid block diagram. Express channels connect the last row with the first row

Grid Processor and demonstrates how programs are mapped onto it. Section 3 characterizes certain aspects of program behavior indicating that existing applications are amenable for execution on the Grid Processor. Section 4 describes related work pertaining to wide-issue and dataflow oriented machines. Section 5 concludes with a discussion on some of the secondary advantages including power reduction and speculation control as well as the remaining issues to be solved.

2 Architecture

The Grid Processor consists of a computation substrate that is configured as a two-dimensional grid of fine-grained computation nodes connected by an interconnection network. The compiler partitions the program into a sequence of blocks (basic blocks or hyperblocks [16]), performs renaming of temporaries, and schedules instructions in a block to nodes of the grid. Instruction traces generated at run-time could be used instead of blocks generated by the compiler. Blocks are fetched one at a time and their instructions are mapped to the computation nodes *en masse* as assigned by the compiler. Execution proceeds in a dataflow fashion with each instruction sending its results directly to other instructions that use them. A set of interfaces are used by the computation substrate to access external data.

2.1 Computation Nodes

Figure 1 shows a high level overview of the grid with some of the associated interfaces. Nearby neighbors in the grid are connected by short wires that have small communication delays [1]. Fast express channels connect nodes that are physically far apart in the grid. The instruction sequencer fetches blocks of instructions from the instruction memory and places those instructions on the nodes as scheduled by the compiler. The block termination control interfaces with the register file and the memory interface, detects when a block completes execution, and commits architecturally visible data to the register file and memory. The memory interface is used to communicate with the load/store queue, caches, and main memory.

[1]The figure shows one possible grid interconnect topology as an example only.

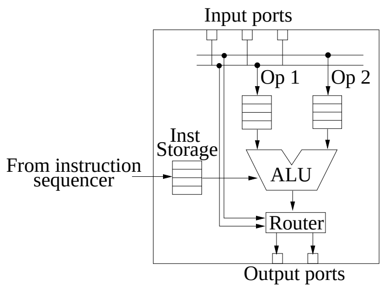


Figure 2: Organization of a computation node

The computation nodes are lightweight units that perform the function of execution, temporary storage, and data forwarding. Each computation node consists of a set of functional units, storage structures, an instruction wakeup unit, a router, and read/write ports for communication. Figure 2 shows the layout of a computation node. The functional units consist of an integer unit and

```
0x0000 add r1, r2, r3    // I1
0x0004 add r2, r2, r1    // I2
0x0008 ld r4, (r1)       // I3
0x000c add r5, r4, 1     // I4
0x0010 beqz r5, 0xdeac   // I5

// End of block B1

0x0014 add r10, r2, r3   // I6
0x0018 add r11, r2, r3   // I7
0x001c ld r4, (r10)      // I8
0x0020 ld r5, (r11)      // I9
0x0024 mul r31, r4, r5   // I10
0x0028 bne r31, 0xbee0   // I11

// End of block B2

0x002c xor r8, r5, 1     // I12
0x0030 sll r9, r4, r8    // I13
0x0034 add r13, r9, 8    // I14
0x0038 add r12, r9, r2   // I15
0x004c sw r13, r12       // I16

// End of block B3

0x0050 add r1, r6, r9
.....
0x0070 jmp 0x0050
```

Figure 3: A sample instruction stream

optionally a floating point unit that perform the actual execution. The storage structures include a set of queues and buffers for storing the instructions, their input operands, and data tokens that need to be forwarded to other nodes in the grid. The instruction wakeup unit matches instructions with their operands as they arrive and issues them to the functional units for execution. The router examines tokens in the storage structures and forwards them along one of the many paths out of this node to their eventual targets. Data tokens meant for other nodes bypass the ALU and are directly forwarded by the router to their destinations.

2.2 Execution Model

The compiler partitions the program into a sequence of *blocks*. Blocks are constructed such that there are no internal control flow changes, and all control transfers out of a block initiate instructions in other blocks. These blocks may be basic blocks, hyperblocks, or program traces generated at run-time. Figure 3 shows a stream of instructions that has been partitioned by the compiler into three different blocks (basic blocks in this case) B1, B2, and B3. Explicit *move* instructions, separate from the computation instructions, are generated for the registers read by every block. The move instructions fetch block inputs from the register file and pass them as internal (temporary) values to the block. Figure 4 shows the Data Flow Graph (DFG) of the blocks B1, B2, and B3 in Figure 3 along with the *move* instructions. As shown in the figure, all instructions have been renamed with temporary registers for their operands and *move* instructions generated for every input register. For example, in block B1, two move instructions, `move t2,r2` and `move t3,r3` are generated by the compiler for input registers r2 and r3. Inside a block, all values are referenced using temporary names. The *move* instructions associates register inputs of the block and temporaries. Data values that must be passed to other blocks are written to the register file.

At run-time, the instruction sequencer fetches a block from the instruction memory and maps it onto the grid *en masse*; there is no serialization of fetch, decode and rename for the instructions within a block. Individual instructions are written to the storage structures of the nodes to which they have been assigned at compile time. Block execution is initiated by the move instructions which read register data and send them to their consumers. The instruction wakeup unit matches incoming data with an instruction and issues ready instructions to the functional unit for execution. The results of the computation are tagged and forwarded by the router through the interconnect to their eventual destinations.

2.2.1 Instruction Mapping

The compiler generates a mapping by physically laying out the data flow graph of each block on a grid. Every computation instruction in the block is assigned to a node in the grid, with the critical path scheduled along the shortest possible physical path. All output operands are renamed with the positions of consumer nodes. Move instructions serve the purpose of associating register data with positions of their consumer nodes. Figure 5 illustrates a layout of the grid with instructions mapped on the computation elements. Instructions I1 and I2 of block B1 in Figure 4 are mapped on the grid at positions (0,1) and (1,1) respectively. Correspondingly, the move instruction `move t2, r2` has (0,1) and (1,1) encoded in its destination fields and register name r2 in its input field.

2.2.2 Instruction Wakeup and Execution

As described in section 2.1, multiple instructions are mapped onto a single node and data are written to an operand buffer when they arrive. Upon arrival of a data token, the instruction and operand buffers are examined to wake-up and issue ready instructions. The wakeup delays will be considerably smaller than seen in conventional cores because of smaller issue windows. The computation node serially performs two operations whenever an operand arrives — *wakeup* and *execute*. Serializing wakeup and execute may increase the cycle time along the execute-execute path of dependent instructions. Wakeup-execute can be pipelined into two stages, if the instruction wakeup does slow the clock. The execute phase of the producer can be overlapped with the wakeup phase of its consumer.

Conventional superscalar cores have dedicated bypass paths to forward data which can be used

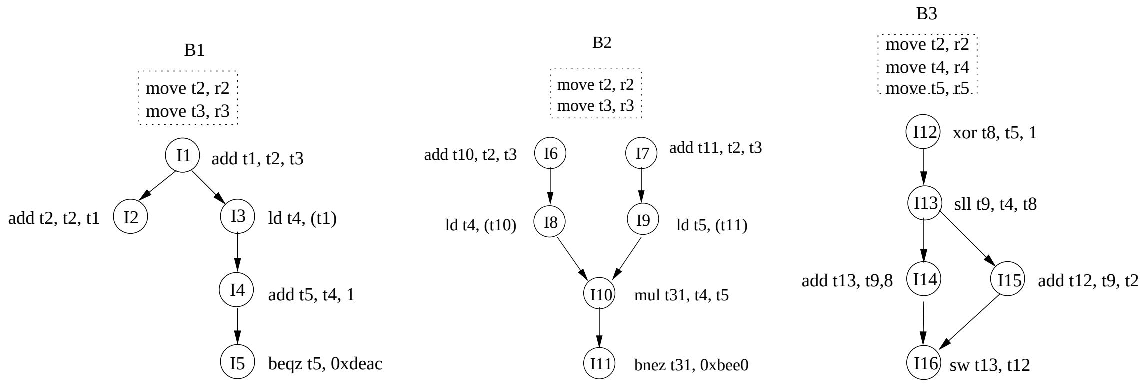
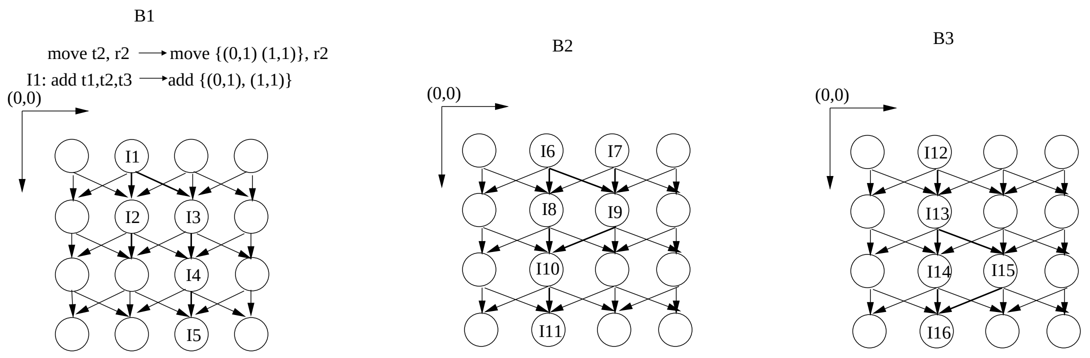


Figure 4: Basic blocks shown as dataflow graphs. Registers are marked with "r" and temporaries with "t".



B1
move t2, r2 → move {(0,1) (1,1)}, r2
I1: add t1,t2,t3 → add {(0,1), (1,1)}
(0,0)
I1
I2
I3
I4
I5
B2
(0,0)
I6
I7
I8
I9
I10
I11
B3
(0,0)
I12
I13
I14
I15
I16

Figure 5: Basic blocks mapped on a grid of dimension 4x4, with the 3 nearest neighbors reachable directly. Instruction destinations are ordered pairs (x, y), which identify a consumer with a relative position x nodes below and y nodes to the right of the producer.

to guarantee that following the execution of an instruction its dependents will have that data in the next cycle. In the Grid Processor, since data are routed dynamically, there is no dedicated path that is guaranteed to be free when an instruction completes execution to forward its data to its consumer. However, there are several mechanisms that can alleviate this problem. Special wakeup tokens could be generated during the issue stage of a producer instruction. They reach the consumer nodes at the end of the stage, reserving a channel for the data to follow in the next cycle. Alternately, speculative instruction issue could be used to hide the select latency with local rollback mechanisms in the event of incorrect issue.

2.2.3 Block Mapping

The instruction and operand storage structures at a node can be used to buffer multiple instructions and data, which are associated through tags. There are three reasons to have multiple instructions mapped on a node. First, graphs larger than the physical grid can be folded over and mapped on the grid with more than one instruction at a node. Second, instructions from different blocks that are fetched speculatively (using control speculation or from speculative threads [19]) can be mapped at a node. Finally, blocks from different threads can also be mapped to support multithreading.

2.3 Instruction Encoding

The Grid Processor ISA is divided into data movement instructions and computation instructions. Data movement instructions include *move*, *split* and *repeat* instructions. The *move* instructions fetch block inputs from the register file and pass them as temporary values to the block. Encoding space limitations restrict the number of targets that can be specified in an instruction. The *split* instructions replicate data to reach additional targets. The range of each target (distance from the producer) that can be specified is finite. The *repeat* instructions are used to forward data to targets outside the range. There is a trade-off between the instruction size, number of specifiable targets and the range of each target.

Every instruction is encoded with an opcode field, destination field, and in the case of *move* instructions, an input field. The destination field consists of multiple targets, with each target encoded with the position of the consumer expressed as an offset. The *move* instructions are encoded with an input register name and its destinations. Figure 5 shows a sample encoding of a move instruction and a computation instruction. The *move* instruction `move t2,r2` is encoded as `move (0,1),(1,1),r2` corresponding to the input register r2 and consumer instructions I1 and I2 that are mapped at (0,1) and (1,1) respectively. Instruction I1 is encoded with destinations corresponding to consumers I2 and I3 of the temporary t1. I2 is mapped at the node directly below I1 and I3 is mapped at the node one to the right and one below I1. An extra bit (not shown in the figure) is necessary to specify the order of the input operands.

2.4 Role of the Compiler

The compiler plays an important role in the Grid Processor. Apart from detecting ILP, the compiler constructs blocks that are scheduled on the grid, and defines mechanisms for intra and inter-block communication. The compiler must also generate data movement instructions to overcome encoding space limitations.

In the Grid Processor, blocks are fetched as a single unit, mapped on the grid, and executed in a dataflow fashion. Since the instructions are fetched at a block granularity, it is desirable to have large blocks and good *block utilization.* Block utilization is defined as the ratio of dynamically executed instructions to the static size of the block. One method of building large blocks is to build hyperblocks based on profiling information. Register file communication for data passed between successively executed blocks can be bypassed using the grid interconnect, thereby "stitching" these blocks as a single dataflow graph. The compiler must define interfaces and mechanisms to stitch together multiple blocks. Since data movement instructions add overhead, when scheduling the graph, the compiler should minimize the critical path and attempt to minimize the number of such instructions.

3 Preliminary Analysis

In this section, we investigate the amenability of existing applications to the Grid Processor and examine few aspects of program behavior that affect performance. Large blocks with a significant number of block temporaries and a few input and output registers are desirable because they have low register file bandwidth. It is also desirable to have large blocks with high utilization to amortize the cost of block fetch and map. The encoding space needed for representing temporaries is determined by the number of targets of an instruction. Fewer average targets per value produced permits a compact encoding. We examine these characteristics in existing applications to determine how well they map onto the Grid Processor.

Benchmark	Average Instructions per block			
	Static Size	Dynamically Executed	Never executed due to early branches	NOPs
gzip	144	77	59	8
mcf	48	35	8	5
parser	29	27	1	1
art	129	125	2	2
equake	57	52	2	3
ammp	124	103	8	13

Table 1: Block utilization

In our experimental analysis, SPEC CPU2000 benchmarks were compiled using the Trimaran [20] tool set. Three floating point *(equake, ammp and art)* and three integer *(parser, gzip and mcf)* benchmarks were selected for analysis. Hyperblocks were generated for these benchmarks using Trimaran's IMPACT compiler with the *train* input set for profiling. All of the benchmarks were simulated using the Trimaran simulator for 500 million instructions with the *ref* input set. We collected dynamic statistics using modifications made to the simulator to track block size profiles and register usage.

3.1 Instruction Behavior

In this section, we examine some aspects of program behavior. Performance is affected by the block sizes in programs and grid configurations. A profile of the dynamic block size was obtained for all the benchmarks to analyze the trade-off of block size with respect to block utilization. Wide grids have better performance at the cost of increased area with fewer nodes having mapped instructions. We analyze this trade-off for three different grid widths.

3.1.1 Block Size

From our analysis of the SPEC CPU2000 benchmarks, we observed that large hyperblocks can be built. Figure 6 shows the dynamic block size profiles for the different benchmarks. For each of the benchmarks, the figure plots the percentage of execution time spent for each dynamic block size as a cumulative distribution function. Dynamic block size is the number of instructions in a block that are actually executed, excluding predicated instructions that are converted to NOPs.

Nearly 70% of the execution time is spent in blocks of size greater than 26 for the integer benchmarks and blocks of size greater than 65 for the floating point benchmarks. Across the benchmarks, the average number of dynamic instructions in a block ranges from 27 to 125.

High utilization percentages are desirable because blocks are fetched and mapped as a single unit. Blocks with poor utilization have a large number of instructions that are fetched and mapped without being executed. In a hyperblock, instructions may not be executed, either because of early exits from the block or because of predicated instructions being converted to NOPs. Table 1 shows the average number of instructions in each block that belong to the categories described above. For four of the six benchmarks, the average utilization is nearly 90% with average static block sizes ranging from 29 to 129. The benchmark *gzip* shows worse utilization than the other benchmarks, as nearly 40% of the instructions fetched are never executed. However, these preliminary results show a potential for building large blocks with good utilization.