A Design Space Evaluation of Grid Processor Architectures

Ramadass Nagarajan Karthikeyan Sankaralingam Doug Burger Stephen W. Keckler

Computer Architecture and Technology Laboratory
Department of Computer Sciences
The University of Texas at Austin
cart@cs.utexas.edu - www.cs.utexas.edu/users/cart

Abstract

In this paper, we survey the design space of a new class of architectures called Grid Processor Architectures (GPAs). These architectures are designed to scale with technology, allowing faster clock rates than conventional architectures while providing superior instruction-level parallelism on traditional workloads and high performance across a range of application classes. A GPA consists of an array of ALUs, each with limited control, connected by a thin operand network. Programs are executed by mapping blocks of statically scheduled instructions to the ALU array and executing them dynamically in dataflow order. This organization enables the critical paths of instruction blocks to be executed on chains of ALUs without transmitting temporary values back to the register file, avoiding most of the large, unscalable structures that limit the scalability of conventional architectures. Finally, we present simulation results of a preliminary design, the GPA-1. With a half-cycle routing delay, we obtain performance roughly equal to an ideal 8-way, 512-entry window superscalar core. With no inter-ALU delay, perfect memory, and perfect branch prediction, the IPC of the GPA-1 is more than twice that of the ideal superscalar core, achieving an average of 11 IPC across nine SPEC CPU2000 and Mediabench benchmarks.

1 Introduction

Microprocessor performance has improved at a rate of 50-60% per year over the past two decades. In the 1970's, wider datapaths and hardware support for memory management contributed to most of the performance improvement. In the 1980's, microprocessors benefited from levels of integration that allowed mainframe techniques to fit on a single chip: memory hierarchies, speculation, and superscalar execution. Since then, however, the bulk of performance growth has come from faster clock rates. Despite copious research efforts, instruction-level parallelism has improved much less than the clock in actual products; current high-end superscalar processors typically sustain one instruction per cycle, and often much less. Comparatively, through the 1990's, four-fifths of performance growth came solely from faster clock rates: a 40% annual increase from 33MHz in 1990 to over 2GHz in 2001.

Clock rate improvements have come both from technology scaling and deeper pipelines, but more from the latter, with pipeline depths increasing by nearly a factor of four over the last decade. This growth will soon end, as deeper pipelines reach limits on the number of gates per pipeline stage [1]. Once that limit has been reached, clock rates will increase at best with gate speeds, which are estimated to improve at a rate of 12-19% per year [22]. Further performance improvements must come from higher levels of instruction- and thread-level parallelism.

Increasing wire resistance will make achieving higher ILP in conventional architectures more difficult than today. Agarwal *et al.* estimate that the latency to transmit a signal across one dimension of a 35nm chip will be approximately 30 clock cycles, even with optimal repeater placement [1]. In addition to limiting the number of devices useful to a conventional core, the wire delays will make memory-oriented microarchitectural structures slower, making it difficult to sustain even current levels of ILP. Slow instruction issue windows, rename tables, branch predictors, bypass networks [17], register files [13], and caches [12] will reduce IPC for a given clock at feature sizes under 100 nanometers. These issues have already become first-order design constraints. For example, the Alpha 21264 uses clustered functional units and a partitioned register file to overcome wire delays, while the Intel Pentium 4 devotes two pipeline stages solely for routing information — instruction distribution and delivery of values to the register file.

Future microprocessors must thus achieve ILP considerably higher than today's designs, even while being partitioned, and do so with a high clock rate. These future processors must exploit increased device counts to meet the above goals, but must do so while considering the increased communication delays and partitioning requirements [25].

In this paper, we introduce a class of architectures intended to address these problems faced by future systems. Grid Processor Architectures (GPAs for short) are designed to enable both faster clock rates and higher ILP than conventional architectures, even as devices shrink and wire delays increase. The computation core of a GPA consists of a two-dimensional array of nodes, each containing a small instruction buffer and one execution unit. These fine-grained computation nodes are connected using a dedicated communication network for passing operands and data, and are controlled by a single thread

of control that maps large blocks of instructions to the nodes *en masse*. This organization eliminates the centralized instruction issue window and converts the conventional broadcast bypass network into a routed point-to-point network. Similar to VLIW architectures, a compiler is used to detect parallelism and statically schedule instructions onto the computation substrate, such that the topography of the dataflow graph matches the mapping. However, instructions are issued dynamically with the execution order determined by the availability of input operands.

In a GPA, few large structures reside on the critical execution path, enhancing scalability as wire resistance increases. Out-of-order execution is achieved with greatly reduced register file bandwidth and with no associative issue window or register rename table. Compiler-controlled physical layout ensures that the critical path is scheduled along the shortest physical path, and that banked instruction caches reside near the units to which they will issue instructions. Finally, large instruction blocks are mapped onto the nodes as single units of computation, amortizing scheduling and decode overhead over a large number of instructions.

In a GPA, the register file bandwidth is also reduced. Our experiments show that register file writes are reduced by 30% to 90% using direct communication between producing and consuming instructions. On a set of conventional uniprocessor (SPEC CPU2000 and Mediabench [14]) benchmarks, our simulation results show IPCs of between one and nine, running on a substrate that can likely be clocked faster than conventional designs and that will scale with technology. Assuming small routing delays, perfect memory and perfect branch prediction, the GPA *averages* eleven instructions per cycle across these benchmarks.

The remainder of this paper is organized as follows. Section 2 describes the block-atomic execution model of programs on GPAs and how programs are mapped onto them. Section 3 describes the GPA-1, one possible implementation of a Grid Processor Architecture. Section 3.3 presents experimental results that both characterize pertinent aspects of program behavior and show potential and actual performance gains. Section 5 discusses design tradeoffs and extensions to the GPA class of machines. Section 6 describes related work pertaining to wide-issue and dataflow-oriented machines. Finally, Section 7 concludes with a discussion of the strengths and weaknesses of GPAs and plans for future work.

2 The Block-Atomic Execution Model

The execution model implemented on Grid Processor Architectures treats groups of instructions as an atomic unit for fetching, mapping onto the execution resources, and committing. The execution substrate is a collection of ALUs, each of which is architecturally visible and named. For simplicity in this paper, we assume that all ALUs are homogeneous and can execute any instruction.

2.1 Instruction Groups

In the block-atomic execution model, instructions are placed into groups by the compiler. A group has no internal transfers of control; taken branches (and the last instruction in a group) transfer control to a succeeding group. A group could thus be a basic block, a predicated hyperblock [16], or a run-time trace [21].

Data used and consumed by a group are of three types: (1) *group outputs*, which are values created within the group and used by subsequent groups, (2) *group temporaries*, which are values that are produced and consumed within the group, and (3) *group inputs*, which are values produced by preceding groups and must be read when the execution of the group begins. Under block-atomic execution, group temporaries can be forwarded directly from producers to consumers, without ever being written back to any central storage. Group outputs, however, must be written to a central storage like a register file when the group commits. The output of control transfer instructions which specify the address of the succeeding group are also treated as group outputs. Modifications to memory are maintained in a temporary storage until the group is committed.

2.2 Group Execution

The compiler statically assigns each instruction in a group to one of the named ALUs, and no ALU is assigned more than one instruction. Special `move` instructions, used to read group inputs, are assigned to the register file. Execution of an instruction group proceeds as follows: A group is fetched and mapped onto the ALUs in the execution substrate at once. Each instruction in the group is stored in the instruction buffer at the ALU (similar to a reservation station) to which it was statically assigned. The `move` instructions issued at the register file read group inputs and forward the values to the appropriate ALUs.

When all of an instruction's operands have arrived at an ALU, the instruction is executed. This data-driven execution model is similar to that of a traditional dataflow machine [2, 9]. When the instruction completes, its result is forwarded to the ALUs holding consuming instructions, and/or to the register file if the result is a group output.

The physical destinations of the operation's result are encoded explicitly into an instruction. Each destination is referred to by the name of the ALU, so that the result can be sent directly to the target instruction. Operands are delivered directly from producers to consumers (point-to-point) in the grid network rather than being broadcast to all ALUs. Since all operands are forwarded to the location where instructions are buffered, an instruction does not encode the source locations or register names of its inputs, only its outputs.

When all of the instructions in a group have completed, the group is *committed*–group outputs are written back to the register file and updates to memory are carried out. Subsequently, the group is removed from the ALUs, and the next group is mapped onto the execution substrate. In the event of an exception being raised by any instruction in a group, the entire group is re-executed after the the exception is serviced. Some implementations of the execution model overlap both fetch, mapping, and execution of the subsequent group with the execution of the current group.

Figure 1 shows an example of the mapping and execution

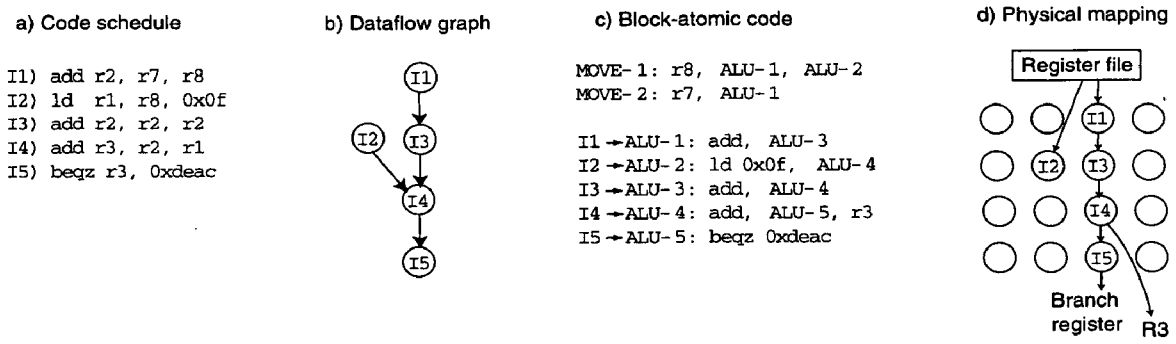

Figure 1: Simple Example of Block-Atomic Mapping

of a group onto a GPA. Part (a) depicts the original code fragment for a basic block consisting of five instructions and part (b) shows the corresponding dataflow graph (DFG). Group inputs are `r7` and `r8`, while `r3` is the group output. The block is scheduled for five ALUs, named ALU-1 through ALU-5, as shown in part (c). Part (d) shows the physical mapping on an execution substrate, consisting of a 4x4 array of ALUs. The two `move` instructions shown in part (c) are mapped at the register file. They read the group inputs and forward them to the consumers at ALU-1 and ALU-2. Temporaries are produced by the instructions at ALU-1, ALU-2, ALU-3, and ALU-4 and are not written to the register file. The output of I4 is a group output and is written to the register file. The branch instruction I5 implicitly writes its result to a special *branch* register, which is used to transfer control to the next hyperblock.

2.3 Key Advantages

The block-atomic model will be effective if the number of instructions in the group is large enough to yield long dependence chains that can benefit from the ALU chaining in the grid. The experimental results in Section 3.3 show that compiler-generated group sizes are significant, when predication is used to eliminate control flow hazards.

This execution model addresses several of the challenges for microprocessor performance scaling described in Section 1. There are fewer large structures involved with the execution: there is no centralized, associative issue window, no register renaming table and there are fewer register file reads and writes. Despite the lack of these structures, instructions can execute in dynamic order, without expensive hazard checking or a broadcasting bypassing and forwarding network that scales poorly with increasing execution width [17]. Furthermore, if the physical instruction layout corresponds to the dataflow graph, communication from producers to consumers will take place along short, point-to-point wires. Instructions off of the critical path can afford longer communication latencies between more distant ALUs. The physical layout of ALUs is exposed to the instruction scheduler, so that the wire and communication delays can be used to help the scheduler minimize the critical path. In the next section, we describe one implementation of a GPA that realizes this block-atomic, data-driven execution model.

3 A GPA Implementation

In Figure 2a, we show a high-level diagram of the GPA-1, our first Grid Processor Architecture design. ALUs are arranged in an *m* by *n* array, shown as 4-by-4 grid in the example. In this implementation, instructions are delivered by instruction cache banks on the left side of the array. The block sequencer and block termination control determines which instruction groups to map to the grid and when each group has been completed and can be committed. Instruction group inputs are fetched from the register file banks and injected from the top of the grid. Operands are passed from producer to consumer instructions through a lightweight network, shown as a mesh augmented with diagonal channels. Memory accesses are routed to the primary cache banks located on the right side of the grid through a separate network.

The architecture of a grid node is shown in Figure 2b. Note that in this terminology, a *node* refers to a functional unit with the logic shown in the figure, rather than a full-fledged processor with its own program counter. Each node contains input ports for arriving operands, instruction and operand buffers, and a router that delivers values to the output ports and the grid network. The buffers hold instructions and input operands until all operands have arrived and the instruction can execute. The router can deliver both values produced by the node's ALU and those being routed through the node to a destination elsewhere in the grid.

The instruction and operand buffers each have multiple entries, enabling multiple instructions to be mapped to a single physical node. A *frame* consists of a single instruction slot in all of the grid nodes and can be pictured as a single virtual grid. Thus each additional slot provides another frame and a virtual grid of nodes. For scheduling and dynamic data forwarding purposes, the (x,y) coordinate of the grid node in the array along with the slot label is used as the name of the destination. In this example, one frame consists of 16 instructions, using one instruction slot in each of the 16 grid nodes. An *8x8* grid with 8 frames would thus be capable of mapping a total of 512 instructions at a time. Groups larger than one frame are allowed to span multiple frames. Free frames can be used to map speculatively fetched groups. Below, we describe the other major features of our design.

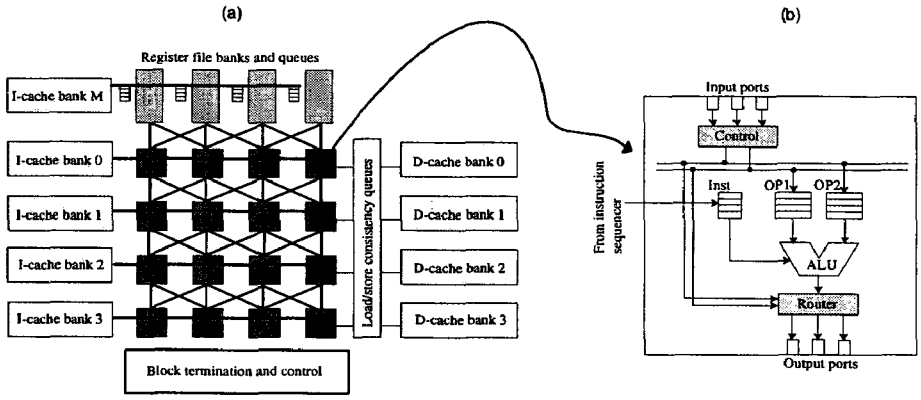

Figure 2: High-level Grid Processor organization

3.1 Instruction fetch and map

Each group of instructions mapped to the grid consists of one predicated hyperblock. These hyperblocks have a single point of entry, and may have multiple exits, but have no internal transfers of control. The primary instruction cache consists of multiple banks, in which one bank is associated with each row. When a hyperblock is mapped onto the grid, each bank reads a row's worth of instructions and delivers those instructions horizontally into the grid along the instruction distribution wires, taking four cycles. After a hyperblock is mapped, branch and target predictors in the block sequencer predict the succeeding hyperblock, and begin fetching and mapping it onto the grid prior to the completion of the previous hyperblock.

3.2 Instruction execution

At the top of the grid resides the register file, which in this example contains one three-ported bank per column. When a hyperblock is mapped onto the grid, the corresponding `move` instructions are fetched and delivered to queues at the appropriate register file banks. Each bank can issue two `move` instructions per cycle, injecting two operands into the grid. The `move` instructions contain the register number to be read and the location of up to three target ALUs within the grid.

When an operand arrives at the node, the control logic attempts to wakeup, select, and issue the instruction corresponding to the frame identifier of the arriving operand. If all of the operands are present, the instruction is issued to the ALU. Upon completion, its result is sent to the output router with the frame identifier and the address of up to two target ALUs. If no new operand arrives at the node in a given cycle or if the instruction whose operand arrived must wait for more operands, any other *ready* instruction is selected and issued.

Operand routing: Because the physical locations of consumers are explicitly encoded within producer instructions, there is a trade-off associated with the instruction fanout. If instructions encode a large number of target consumers, each instruction may be overly large. Too few targets results in extra instruction overhead to replicate the values within the grid. In this example, we support three consumers per instruction, since our results show that over 70% of producer instructions have three or fewer consumers. If an instruction has more than three consuming instructions for a particular value, a data movement instruction called a `split` instruction can be inserted into the schedule to forward results to multiple consumers.

Inter-node network: Four kinds of delays inhibit back-to-back execution of instructions in consecutive cycles: a) routing delays, b) transmission/wire delays, c) instruction wakeup delay and d) delays induced by contention for the wires/ports at the nodes. For instructions on the critical path all these delays should be minimized. From a sensitivity analysis of these features using simulation results, we discovered that the amount of contention in the grid is not substantial and two I/O ports at each node is sufficient. The router and wire delays, however, are the single most important factor in overall performance of the GPA-1.

3.3 Hyperblock control

To increase instruction group size and reduce the number of branches in the program, the GPA-1 uses hyperblocks that include predication and multiple exit points. Hardware support is provided to execute these hyperblocks, which contain predicated instructions and early block exits.

Predication: There are several possible strategies for handling predication within the block-atomic execution model. The GPA-1 uses an *execute-all* approach, in which both predicate paths execute, but only one path delivers a result to the common instructions after the predicate. This approach is implemented by predicating only the leaf instructions in the DFG of the predicated region. This strategy reduces the number of instructions to which the predicate must be delivered and permits execution of all but the leaf instructions of the DFG before the predicate is calculated, reducing the critical path. A special class of instructions called *cmove* (conditional move instructions) are used to implement predication. They accept one input operand and a boolean condition and create one output value. The *cmove* instructions are of two types: `cmove_t` and `cmove_f`. The `cmove_t` instruction forwards

a) Code schedule

```
I1) add r2, r7, r8
I2) ld  r1, r8, 0x0f
I3) cmp p1, R1, #0
I4) ld r9, r8, 0x101f  (p1)
I5) add r2, r2, r9     (p1)
I6) add r3, r2, r1
I7) beqz r3, 0xdeac
```

b) Rescheduled code

```
I1) add  t2, r7, r8
I2) ld  r1, r8, 0x0f
I3) cmp p1, R1, #0
I4) ld r9, r8, 0x101f
I5) add r2,  t2, r9     (p1)
C1) cmove_f r2,  t2, p1
I6) add r3, r2, r1
I7) beqz r3, 0xdeac
```

c) Dataflow graph

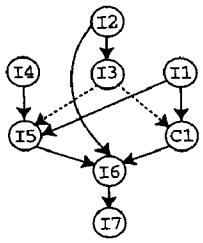

Figure 3: Code example for a Grid Processor

the input operand to the output if the condition is true, while the cmove_f forwards the input if the condition is false. If the boolean condition is not met, then no output is produced.

Figure 3a shows a predicated sequence of code in which instructions I4 and I5 form a predicated region. I6 uses register value r2 which is defined by I1 and conditionally redefined by I5 depending on the predicate p1. Figure 3b shows the rescheduled code with one cmove_f instruction (C1) inserted. C1 is added to ensure that exactly one value for r2, produced either by I1 or I5, reaches I6. The dataflow graph for the rescheduled code is shown in Figure 3c. The data dependences are shown in solid lines and the predicate values passed in dotted lines. Note that the predicate value p1 is now sent only to I5 from the original region, and not I4, since I4 produces a temporary that is created and destroyed in the predicated region (*i.e.*, I5 is the only leaf node in the predicated region).

We discuss other alternatives to support predication in Section 5. The approach taken here is chosen for high performance—few instructions must wait for the predicate to be calculated. However, this approach results in a less efficient use of power; other approaches may be preferable in a power-constrained implementation.

Early exits: A branch from the middle of a hyperblock is called an *early exit*. When a hyperblock contains an early exit branch, the GPA-1 must ensure that only the correct values are ultimately written back to the register file and memory. Further, to maintain program correctness, branch instructions should be executed in serial order. The GPA-1 uses predication to enforce this sequentiality when natural data dependences do not. Every branch instruction is predicated on the complement of the condition for the immediately preceding branch in that hyperblock — this branch should be executed only if the previous branch was not taken.

Like predication, early exits introduce the potential for the same output register name to be produced at several points in a hyperblock. The GPA-1 should guarantee that exactly one value reaches the block termination control. Every such output instruction is predicated on the condition for the immediately following branch — if the branch is taken, this output should be written out, otherwise it should be ignored. Extra predication is necessary only when the same register name is to be produced by multiple instructions in the block and not for every output instruction.

Since the mapped blocks execute in dataflow order, instructions generating output values may execute before a prior taken branch. These results must be filtered so that they do not modify the register file or memory. An index number assigned to each instruction, which indicates its position in static program order, is used to filter values at the block commit logic. When a branch executes, and sends its target to the global control, all outputs generated by instructions later than the branch are discarded by the block commit logic.

Block commit: GPAs benefit from distributing execution state, but that same distribution makes decisions about global control more complicated. The hyperblock can be committed when all stores and output register values have been produced. In the presence of early exits and predication, detecting when all values have been produced requires additional logic at the block termination control. The GPA-1 employs a count of output values statically associated with each hyperblock. When a store or register output fires, it send a signal to the commit logic that sums the signals to detect block completion. If an output is produced by an instruction whose predicate is false, a signal is still sent to the commit logic, but without an associated value. This policy means that a block cannot be committed until all instructions have fired, even those on false predicate paths or those after taken branches. We are investigating specialized networks and in-grid combining trees to ease that restriction.

Block stitching: Thus far, we have described a GPA design in which fetch, map, and execute are serialized across different hyperblocks. However, fetching, mapping, and execution can be overlapped to utilize the available frames and functional units in the grid more effectively. While one hyperblock is executing, the next block can be speculatively fetched and mapped using the address returned by a block level predictor.

Concurrent execution of multiple hyperblocks is also feasible, similar to previously proposed speculative threads [24]. Register values passed between concurrently executing blocks are communicated through the in-grid network, bypassing the register file. Register inputs of speculatively mapped blocks that are not produced by a previously mapped block are delivered to the consumers by executing the corresponding move instructions. We call this mechanism *block stitching*.

Name	Block size		Register usage				Branch exits	Memory conflicts(%)
	Static	Dynamic	Inputs	Temp reads	Temp writes	Outputs		
adpcm	51.1	30.7	6.5	26.7	22.4	5.7	5.7	0
dct	187.9	172.1	14.7	199.8	163.6	7.3	1.7	0
mpeg2	109.7	94.1	11.9	98.5	84.4	7.4	3.4	1.6
gzip	66.2	37.1	11.6	29.8	28.4	7.1	4.4	9.3
mcf	40.1	28.8	4.7	33.6	25.7	2.3	1.7	0.9
parser	20.3	16.2	3.4	13.7	12.7	2.8	1.8	4.8
ammp	82.5	72.3	6.6	67.8	58.1	3.8	8.7	0.6
art	82.7	79.6	25.6	54.2	54.0	20.0	5.3	54.9
equake	50.2	44.1	10.0	34.4	32.7	9.5	2.3	11.9

Table 1: Program characteristics

3.4 Memory access

In the GPA-1, the primary data cache resides on the right-hand side of its execution array. When a load executes at a node, it forwards its effective address and targets from the node to the data cache, which performs the memory access and sends the result of the load directly to the targets. Maintaining the correct ordering between loads and stores that do not have an explicit and visible data dependence between them is a well-known problem. We use traditional load-store queues to maintain sequential memory semantics for loads and stores that arrive out of order. However, since the queues are on the critical path to memory, they may slow accesses down in a wire-dominated design, even if no ordering violations are detected.

4 Evaluation

This section presents a study of pertinent program characteristics, followed by an evaluation of GPA-1 performance across a set of nine applications. We chose three SPEC CPU2000 floating-point benchmarks (*equake*, *ammp*, and *art*), three SPEC CPU2000 integer benchmarks (*parser*, *gzip*, and *mcf*), and three Mediabench benchmarks (*adpcm*, *dct*, and *mpeg2enc*) [14] for our analysis. All benchmarks were compiled with the Trimaran tool set [26] to generate the necessary hyperblocks. The SPEC benchmarks were compiled with the *train* input set and run using with *ref* input set, while Mediabench applications were compiled and run using the same input set. All compilations were performed with full optimizations (*-O4*).

To characterize the applications, the Trimaran simulator–which performs functional execution of the Trimaran-generated code–was modified to track block size profiles and register usage. We collected dynamic statistics for first 1 billion instructions executed in each benchmark.

We used a custom instruction scheduler that accepts the Trimaran-generated hyperblocks as inputs, inserts overhead instructions, and schedules the blocks onto the GPA-1. The scheduler assigns instructions to nodes in the grid using a greedy critical path scheduling strategy. This strategy schedules one instruction per node, with the longest path in the DFG on the shortest possible physical path in the grid. Further, for best proximity to the caches, load instructions are placed as far to the right of the grid as possible. Finally, we assume full floating-point and integer units at each node, so each node in the GPA-1 can execute any instruction.

We estimated performance using a custom event-driven timing simulator. We modified the functional front-end of the Trimaran simulator to generate an execution trace of a hyperblock, which is used by our timing back-end to simulate execution on the GPA-1. The simulator accounts for dynamic behavior including routing latencies, contention for the wires in the grid and at the input and output ports of each node, a memory hierarchy with two levels of data caches and main memory, and next-hyperblock prediction. We fast-forwarded through the first 500 million instructions of each application, and then simulate the following 200 million instructions to obtain timing results.

4.1 Application Characteristics

In Table 1, we display the characteristics of the benchmarks compiled with the Trimaran compiler. The average number of instructions per hyperblock statically produced by the compiler is shown in column 1 and the average number of instructions dynamically executed per hyperblock is shown in column 2. These sizes correspond to only the useful instructions in a hyperblock; overhead instructions (`move`, `split`, and `cmove`) are not included for the static sizes and further, instructions that receive false predicates and those beyond a taken early exit are not included in the dynamic sizes. Unsurprisingly, the integer SPEC CPU2000 benchmarks show the smallest dynamic hyperblock sizes, ranging from 16 to 37 instructions on average.

The next four columns show the number of register inputs to each hyperblock, the number of temporary reads and writes, and the output values. The temporary reads and writes are values produced, forwarded and consumed within the grid and require no register file accesses. A significant reduction (30% to 90%) in register file bandwidth is achieved. Some of that reduction may be lost, however, if large hyperblocks are split to fit onto a GPA of finite size.

The branch exits column shows the average number of branches per hyperblock. Larger numbers of potential early exits will require complicated control to pass the correct register values to the register file, and to subsequent blocks if group stitching is supported.

Finally, the rightmost column shows the fraction of hyperblocks that contain a store with a later load to the same address. The number is significant (greater than 5%) for three bench-

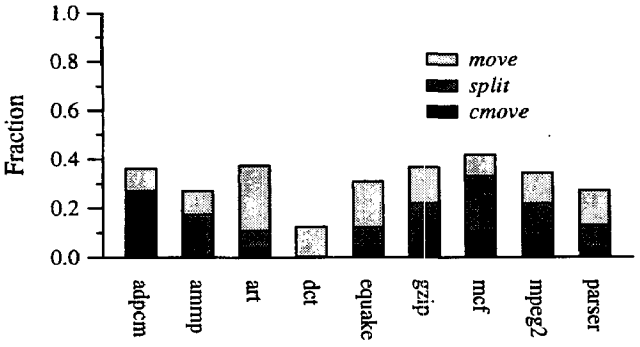

Figure 4: Overhead of Block-Atomic Execution

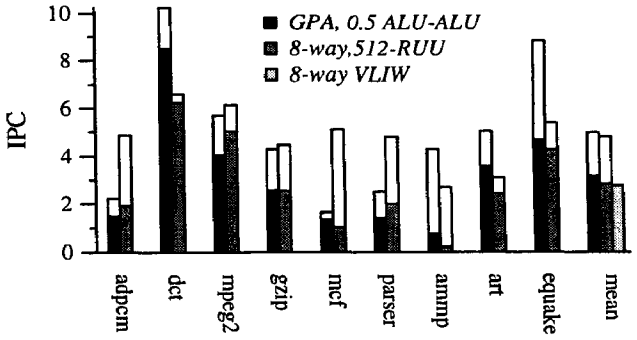

Figure 5: Performance Comparison of GPA with superscalar

marks, indicating that even with greatly reduced register spills due to the large number of compiler-visible, intra-hyperblock temporaries, some mechanism to reduce the penalty or frequency of load-store conflicts, such as dependence speculation, will likely be needed.

Figure 4 shows the fraction of overhead instructions (`move`, `split`, and `cmove`) required by the block-atomic execution model. `split` instructions are added when an instruction requires more than three targets. The average across the benchmarks is approximately 35% of all instructions. However, only the `split` and `cmove` instructions add execution overhead by consuming instruction slots on a grid; the `move` instructions are kept in a separate instruction cache bank, and do not consume execution resources, nor do they reside in between dependences on the critical path. Discounting the `move` instructions, which only affect program binary size and lower-level cache miss rates, the remaining overhead instructions consume less than 20% of the instructions scheduled on the grid.

4.2 Performance evaluation

The baseline GPA-1 configuration is an 8x8 grid with 32 frames. Each node is connected to three of its neighbors in the next row. *Express channels*, which are higher-level metal channels route operands from the bottom to the top of the grid, at double the velocity of the short node-to-node wires. In this simulated implementation, integer add and logical instructions require a single cycle to execute. Other instruction latencies are configured to be similar to that of the Alpha 21264. We assume that long latency operations such as floating-point adds and multiplies can be fully pipelined.

We evaluate performance on GPA-1 configurations with both perfect and realistic assumptions for memory and next hyperblock-prediction. When modeling a realistic memory system, we simulate a memory hierarchy with 64KB, 2-way L1 data caches with 3-cycle access, and a 1MB, 4-way, 13-cycle L2 cache, a 62-cycle physical memory latency. When modeling realistic branch prediction, we simulate a 2-level global branch predictor with a 14-bit history, a 16K entry pattern history table, and a 512-entry, 2-way branch target buffer. We assumed a perfect instruction cache for all experiments.

Our GPA simulator does not model wrong-path execution. When a misprediction occurs, we account for the delay by stalling the mapping of the subsequent hyperblock until its target is known, permitting correct control flow to resume. We also assume ideal behavior for store-load pairs with the same address: those loads are stalled until the store completes, whereas independent loads are allowed to issue when ready. This assumption is less optimistic than it would be in a conventional system, as discussed in Section 5.

The major features that influence performance in the GPA-1 are the organization of the ALUs (the number of ALUs and their layout), the interconnect network latency, and the number of I/O ports at each node. Our default GPA-1 configuration assumes an 8x8 grid of ALUs with two I/O ports at each node. The interconnect network is configured such that the three nodes directly below any given node can be reached in a single hop. Two components constitute the routing delays of passing an operand from a producer to its consumer: wire delay, which depends on the physical distance in nodes between a producer and consumer, and the router delay at every node in the path. The default GPA-1 parameters assume that the wire and router each consume a quarter cycle, totaling half a cycle per hop of routing delay.

Comparison with alternate architectures: Figure 5 shows a direct performance comparison of the 8x8 GPA-1 to an idealized 8-way issue superscalar processor. The left bar in each benchmark cluster shows the performance of the GPA-1, while the right bar shows the same for the superscalar processor. The white portions of each bar represent the IPC assuming perfect memory and perfect branch (or next-hyperblock) prediction. The colored portions show the IPC assuming realistic memory and realistic branch prediction.

The superscalar processor was simulated with the SimpleScalar tools [4], for which we assumed a 512-entry RUU (instruction window and reorder buffer). Furthermore, we assumed that the clock rates of both the machines were the same, despite the difficulty of building an 8-wide, large window superscalar core, with full bypassing, that could be clocked at the same frequency as the distributed-window GPA-1. We also simulated each benchmark on Trimaran's VLIW processor simulator assuming 8-way issue, perfect memory, and static branch prediction. In Figure 5, the third bar in the last cluster shows the mean performance of this VLIW machine. Despite the assumption of perfect memory, the simulated VLIW processor performs worse on average than the superscalar proces-

Name	Perfect Mem + BP		Realistic Mem + BP	
	IPC No stitch	Stitching Speedup	IPC No stitch	Stitching Speedup
adpcm	1.4	1.4	1.1	1.2
dct	4.0	2.5	3.7	2.2
mpeg2	3.0	1.8	2.7	1.4
gzip	1.9	2.2	1.4	1.7
mcf	1.0	1.5	0.6	1.9
parser	1.0	2.4	0.8	1.7
ammp	2.3	1.8	0.2	3.1
art	2.5	1.9	1.5	2.2
equake	2.5	3.4	2.0	2.2

Table 2: Speedup achieved by stitching

Name	Average hops per data value		
	Input	Temporary	Memory
adpcm	2.8	1.8	1.9
dct	2.9	2.4	3.7
mpeg2	3.3	1.5	3.8
gzip	3.0	2.1	2.7
mcf	2.1	2.8	1.9
parser	2.6	1.8	1.8
ammp	2.4	1.7	4.9
art	3.8	1.7	3.2
equake	3.8	2.0	2.8

Table 3: Average hops for different types of data operands

sor with realistic memory.

The results show promise for the GPA-1. In four of nine benchmarks with perfect memory and branch prediction, and for five of nine benchmarks with the more realistic models– the GPA-1 demonstrates superior performance to the *idealized*, large-window superscalar engine, which in turn showed higher performance, in every case, than the VLIW core. The disparity is higher for perfect memory and prediction, indicating that the GPA-1 has higher performance potential, but that further improvements in the memory system and branch predictor are more important for GPAs than for conventional superscalar cores. The code for which the GPA-1 performs best is *dct*, showing 10.2 IPC for perfect memory/prediction and 8.5 for the realistic assumptions. This architecture is able to harvest substantial ILP when it exists. Of the benchmarks on which the GPA-1 performs substantially worse than the superscalar processor, three of them (*adpcm*, *mcf*, and *parser*) are due to disparities in the compilers: the Compaq C compiler (V6.3-025) with full optimization versus the lower-performing Trimaran compiler. Individual analysis of these three benchmarks showed that, the superscalar core with perfect memory and predictors achieves higher IPC than there is available ILP in the Trimaran-generated code, assuming an ideal machine with infinite resources. Extensions to our compiler infrastructure to handle small inner loops with loop-carried dependences will improve the performance of those benchmarks on the GPA-1.

Block stitching: Concurrent block execution on the GPA-1 efficiently utilizes available frames and functional units, and benefits from block stitching. Table 2 shows the speedup obtained due to stitching. The second and third columns correspond to the baseline GPA configuration with perfect memory and prediction assumptions and the last two columns correspond to the GPA configuration with realistic assumptions. The second and fourth columns show the IPC achieved on a GPA without stitching. Columns 5 and 6 show corresponding speedups with stitching. Block stitching provides roughly a factor of two speedup for both perfect and realistic assumptions. These results indicate that the ability to map multiple blocks, even speculatively, to a GPA is critical for competitive performance of sequential threads.

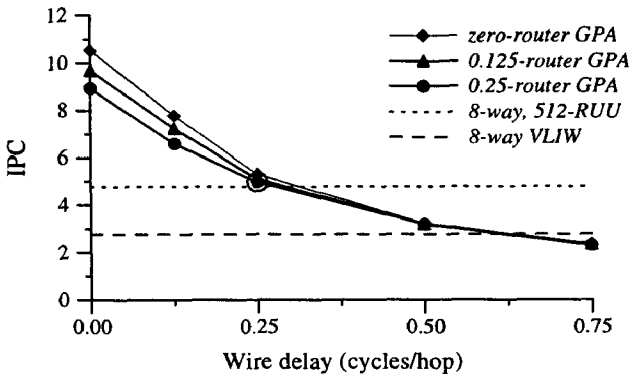

Figure 6: Sensitivity to wire delays

Routing delay: Our results show that the routing delay from an operand's producer to its consumer is the single largest determinant of the aggregate GPA-1 performance. The portions of operand routing delay that we measure or vary in this section are the three most significant components of that total delay: 1) the number of hops traversed, 2) the inter-node wire delay, and 3) the router delay at each hop.

Table 3 shows the average number of hops needed to route data in the grid for the GPA-1. Because the number of input register values is non-negligible, input operands are typically routed between two and three hops in the network. The scheduler is effective at reducing the number of hops needed for temporaries, which require roughly two hops on average. The number of hops to and from memory varies more than the routing of operands, and represents one of the performance bottlenecks that more sophisticated schedulers must avoid.

The inter-node wire and router delay is critical for performance. We simulate GPA-1 configurations for router delays of 0, 0.125, and 0.25 cycles with wire delays varying from 0 to 0.75 cycles. Figure 6 shows the effects on mean IPC (across our nine benchmarks) as the inter-node wire delays are varied, assuming perfect memory and prediction. For example, the bottom curve corresponds to a fixed router delay of 0.25 cycles. The GPA-1 configuration with zero wire and router delays models back-to-back execution in consecutive cycles, with performance being limited only by the availability of functional resources and instruction buffers. The horizontal lines show the mean IPC of the VLIW and superscalar cores with perfect memory/predictor assumptions. The circled dot

Name	GPA-1 IPC	Contention		Connectivity		
		5/5 I/O	Zero	C5	C6	C10
adpcm	2.2	2.2	2.4	2.2	2.4	2.5
dct	10.2	10.0	19.8	10.4	11.0	11.9
mpeg2	5.7	5.8	6.9	5.8	6.7	6.4
gzip	4.2	4.2	4.8	4.2	4.3	4.5
mcf	1.6	1.6	1.7	1.6	1.8	1.7
parser	2.5	2.5	2.6	2.4	2.3	2.5
ammp	4.2	4.3	4.6	5.2	4.6	5.9
art	5.0	4.8	5.4	4.7	5.0	4.9
equake	8.8	8.9	11.5	8.3	9.0	9.6

Table 4: Sensitivity analysis for different features

shows the GPA-1 configuration that was used for the performance results, shown in Figure 5. As the wire delay of the GPA-1 shrinks to zero from 0.5 cycles, close to a factor of two improvement in mean IPC is achieved, from just under five to over nine. When both the router delay and the wire delays are set to zero, the mean IPC is almost 11.

The router operation at the first hop is assumed to be in parallel with instruction execution. As shown in Table 3, the average number of hops per communication is 2. This corresponds to one router and two wires being traversed resulting in an effective communication delay of 0.75 for the baseline GPA. Thus for any GPA configuration, the wire delay affects performance more than the router delay. The wire and router delays are analogous to both operand bypass delays in a conventional superscalar microarchitecture, and inter-cluster delays in a partitioned superscalar or VLIW processor. Our baseline parameters assume a quarter cycle delay each for both the wire and router. Conversely, we assume that in the superscalar processor, dependent operations can issue in back-to-back cycles. It is possible, however, that the point-to-point communication in a GPA can be faster than operand bypassing in future superscalar processors. That analysis is beyond the scope of this paper.

Connectivity: Table 4 presents the effect of connectivity and contention for I/O ports on IPC. All the results presented in the table assume a 8x8 grid with perfect memory and perfect prediction. The second column shows the IPC for the baseline GPA-1 configuration. The third corresponds to the GPA-1 with 5 input and 5 output ports at each node. The fourth column corresponds to the GPA-1 with zero contention for the I/O ports and wires in the grid. Infinite inter-node communication paths improves performance minimally, indicating that contention plays a minor role. To examine the effect of the richness of the interconnect in the grid, we varied the number of nodes (5, 6, and 10) to which each node is connected. These results are shown in the columns C5, C6, and C10 respectively. C5 and C10 correspond to a node being connected to 5 of its neighbors in the next row and next two rows, respectively, whereas C6 corresponds to a node connected to three of its neighbors in the next two rows. The average performance speedup when simulating 10 point-to-point paths per node instead of 3 was 1.1. However, higher connectivity could increase delays in the network due to router complexity and number of wires.

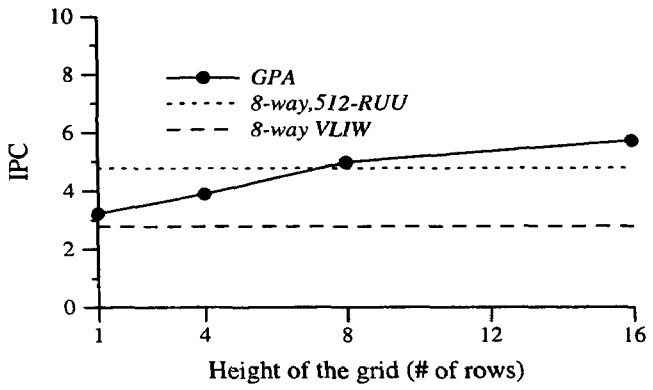

Figure 7: Sensitivity to grid height

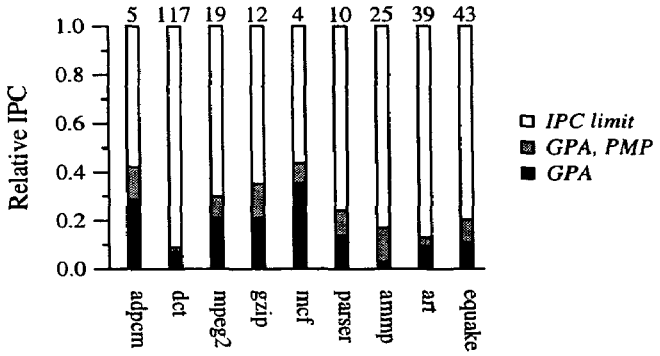

Figure 8: GPA effectiveness

Grid dimension: Figure 7 shows the sensitivity of performance to grid height. Performance increases linearly up to 8 rows, beyond which the performance improvement tapers off. Some benchmarks perform best with 8 rows. The loss in performance when more rows are added is due to the longer routing delays to reach the bottom of the grid — either to reach the termination control or to pass through an express channel. Programs with large available ILP and large block sizes benefit from increase in the number of rows. Among the benchmarks we studied, *dct*, *mpeg2*, *ammp*, and *art* fall into this category.

GPA Effectiveness: Figure 8 shows the fraction of achievable ILP that the GPA-1 exploits in each benchmark, both for perfect memory/prediction and realistic assumptions. Each bar is normalized by dividing the GPA-1 performance by the IPC observed on an ideal machine. The ideal machine is modeled by simply traversing the program dataflow graph, and dividing the number of instructions by the critical path length. That ideal IPC value for each benchmark resides atop its corresponding bar. The middle bar corresponds to the GPA-1 configuration assuming both perfect memory and prediction (PMP), while the bottom bar corresponds to the configuration with realistic assumptions for memory and prediction. We see that the GPA-1 exploits between 10% and 40% of the available ILP in each benchmark.

While these results are promising, they represent only an unoptimized, first-cut GPA design. The performance benefits

are likely to diminish as we add detail to our simulator, but are also equally likely to improve as we enhance the scheduler and tune the performance of the architecture. In the next section, we describe some of the possible extensions and alternatives to the GPA-1 that could further improve performance.

5 Design Alternatives

Because the design space for GPAs is large, there are many unknowns about performance/complexity trade-offs for the new aspects of this system. In this section, we describe unexplored opportunities for performance tuning in the grid network, the block control logic, and the memory system. We then describe more radical extensions to the block-atomic execution model that may eventually provide higher performance or flexibility in the architecture.

5.1 Performance tuning

Grid network design: One of the key determinants of GPA performance is the logic and wire delay between producers and consumers on the critical path. With clock rates close to the speed of the grid ALUs, the latency in the routers and wires in the network must be kept as short as possible. The latency to communicate ALU results from producer to consumer will depend heavily on the distance and the number of hops in the grid network. Larger degree routers will reduce the number of hops, but increase the delay per hop. Achieving the right balance between near and far communication between producers and consumers is critical for high performance. While this trade-off is similar to those found in multiprocessor interconnection networks, the fine granularity of the operand network magnifies the effect. Furthermore, since the routers may not necessarily have dedicated channels and ports, efficient flow control of the operand packets will be critical for obtaining high performance. To reduce the handshaking overhead required by many flow control protocols, we are currently examining techniques for pre-reserving network channels to a consumer node while the producer ALU is executing its instruction. Such techniques are similar to Flit-Reservation Flow Control, which has been proposed for coarser-grained on-chip networks [18]. We are also examining how express channels can reduce the communication latency, by trimming the number of hops between distant producer and consumer ALUs [8], as well as circuit techniques to minimize delays in the routers.

Predication strategies: Since the GPAs run with a data-driven execution model, predication is difficult to implement. The problems with predication are a) communication of predicate bits to instructions in predicated regions and b) added complexity of block termination control to handle instructions that receive false predicates. The simplest of the strategies is to send predicate bits to all instructions in the predicated region. This strategy avoids superfluous execution but requires high bandwidth for predicates. Alternately, predicates may be sent only to the root instructions in the data dependence subgraph controlled by the predicate, reducing predicate fan-out. Both of these approaches limit performance as all instructions in the predicated region need to wait until the predicate bit is received [3]. In our current solution, the compiler predicates only those instructions that update stable storage (stores and register file writes). That strategy provides a critical path reduction, since the predicate computation is not needed until later, plus a lower fanout of the predicate values is needed, at the expense of less efficient use of power.

Memory system: Efficient delivery of instructions into an ALU grid relies upon placing the instructions in the corresponding instruction cache bank to avoid routing delays. Since the schedule may have holes due to unused ALU slots, an uncompressed version is likely to be large. While not discussed above, we anticipate maintaining the program code in a compressed format in the memory hierarchy below the L1 instruction cache to conserve both capacity and bandwidth. In the data memory, the first-order challenge is to maintain proper ordering of load and store instructions to the same memory locations. Our existing design requires a structure similar to a load/store queue with store-load forwarding. We are exploring both speculative and conservative strategies to detect ordering violations and enforce ordering among subsets of the load and store instructions. We are also exploring a scheme in which previously communicating store-load pairs speculatively communicate via point-to-point messages, bypassing the memory system.

5.2 Execution model extensions

Grid speculation: In addition to the speculative block mapping and execution described in Section 3, a GPA can potentially support speculation and inexpensive recovery within a single hyperblock. For example, if the dependence between a load and a store is unknown, a GPA may issue the load speculatively, fetch the data from memory, and pass it to the instructions that will consume the value. If the load is later determined to have needed the result from a prior store, the entire hyperblock need not be nullified. Instead, the grid may be able to employ *selective re-execution* by injecting the new load result into the grid where it is routed to the instructions that are still mapped onto the ALUs. The new values trigger only those instructions along the dependence path from the load to the end of the block for re-execution. This capability will greatly reduce the overheads associated with rollback from a data value misprediction.

Frame management: Multiple frames essentially provide multiple logical processors on the same grid substrate. Managing these frames is key to the execution model implemented by a GPA. As described above, these frames can be used for speculative mapping and execution of hyperblocks in a sequential program. For applications that consist of independent threads of control, the frames can support a multithreaded execution model. A subset of frames may be allocated to each thread and then used in a manner chosen by each thread: sequential with speculative mapping or data parallel with mapping reuse, as described below. It is also possible that different threads could share the same frame, with each thread using a physically different subset of the ALUs, but the complexity for this level of sharing appears prohibitively complex.

ALU control: While Section 3 describes each ALU as having only a single instruction per frame with only data triggered control, we are examining extensions that provides a small amount of additional local control at each ALU site. The extensions include increasing the amount of local instruction and data storage, and treating the ALUs as simple microcontrollers. *Mapping reuse* is possible in the grid by delivering inner kernels to each ALU which could then self-sequence through the instruction blocks with limited intervention from any global grid controller. The potential exists for mapping a repeated hyperblock to a grid once, and then dynamically re-instantiating the block locally, with no refetching, for multiple iterations. Different iterations can thus be executed in different frames.

6 Related Work

The goals of high clock rate and high IPC are not unique to GPAs. Many prior approaches have attempted to use both static and dynamic techniques to discover and execute along the critical path of a program, but they are too numerous to discuss here. In this section we describe what we believe to be the most relevant related work.

Dennis and Misunas proposed a static dataflow architecture [9], and Arvind proposed a Tagged-Token Dataflow architecture with purely data-driven instruction scheduling for programs expressed in a dataflow language [2]. Culler later proposed a hybrid dataflow execution model where programs are partitioned into code blocks made up of instruction sequences, called threads, with dataflow execution between threads [7]. Our approach differs from these in that we use a conventional programming interface with dataflow execution for a limited window of instructions, and rely on compiler instruction mapping to reduce the complexity of the token matching.

In a sense, GPAs are a hybrid approach between VLIW [10] and conventional superscalar architectures. A GPA statically schedules the instructions using a compiler, but then dynamically issues them based on data dependences. Other efforts have attempted to enhance VLIW architectures with dynamic execution. Rau proposed a split-issue mechanism to separate register read and execute from writeback and a delay buffer to support dynamic scheduling for VLIW processors [20]. Grid Processors share many characteristics with the Transport Triggered Architectures proposed by Corporaal and Mulder, including data driven execution, reducing register file traffic, and non-broadcasting bypass of execution unit results [6, 5].

Others have looked at various naming mechanisms for values to reduce the register pressure and register file size. Smelyanskiy *et al.* proposed Register Queues for allocating live values in software pipelined loops [23]. Llosa proposed register sacks, which are low bandwidth port-limited register files for allocating live values in pipelined loops [15]. Patt proposed a Block-Structured Instruction Set Architecture for increasing the fetch rate for wide issue machines where the atomic unit of execution is a block and not an instruction [11].

Many researchers are exploring distributed or partitioned uniprocessor designs. Waingold *et al.* proposed a distributed execution model with extensive compiler support in the RAW architecture [29]. The RAW architecture assumes a coarser-grain execution than does the Grid Processor, exploiting parallelism across multiple compiler-generated instruction streams. Ranganathan and Franklin described an empirical study of decentralized ILP execution models [19]. Sohi *et al.* proposed Multiscalar processors, in which a single program is broken up into a collections of speculative tasks [24].

A different approach to creating a distributed window used dynamic traces for the execution partitions [28]. In that work, Vajapeyam and Mitra proposed renaming temporary registers within a trace to reduce the needed global register file and rename bandwidth; GPAs use a similar approach, except that the renaming is performed statically. Unlike that design, however, the GPA-1 executes hyperblocks in a fine-grain dataflow fashion and overlaps speculative tasks/hyperblocks on the same computation substrate.

Finally, Uht *et al.* are currently investigating an architecture [27] that is also intended to exploit high ILP with many ALUs in a single execution core, but using different communication mechanisms than Grid Processor Architectures.

7 Conclusion

This paper has introduced Grid Processor Architectures as a new class of microarchitectures, that are intended to enable continued scaling of both clock rate and instruction throughput. By mapping dependence chains onto an array of ALUs, conventional large structures such as register files and instruction windows can be distributed throughout the ALU array, permitting better scalability of the processing core. By delivering ALU results point-to-point instead of broadcasting them, GPAs mitigate the growing global wire and delay overheads of conventional bypass architectures. Our initial studies on sequential applications are promising, with the grid processor achieving IPCs ranging from 1 to 9, competitive with those of idealized superscalar microarchitectures, and exceeding those of VLIW microarchitectures.

It it not clear that GPAs will be superior to the conventional alternatives, which may find more incremental, but equally good solutions to the wire delay and clock scaling problems. GPAs have several disadvantages; they force the data caches to be far away from many of the ALUs, and incur delays between dependent operations due to the network router and wires, which can be significant. The complexity of frame management and block stitching (allowing successor hyperblocks to execute speculatively) is significant and may interfere with our goal of fast clock rates.

However, future architectures must be partitioned somehow, and the partitioning and the flow of operations are likely be exposed to the compiler, while still preserving dynamic execution. Many of the techniques discussed herein are thus likely to appear in future designs. We are actively working to refine the microarchitecture of the GPA-1 and the hyperblock scheduler with the anticipation that the hardware complexity can be further reduced without undue burden on the software. Future work will also include an exploration of different grid execution models for mapping streaming/media, scientific/vector, and multithreaded codes.

Acknowledgments

Many thanks to the anonymous referees for their valuable suggestions on an earlier version of this paper. Thanks also to the Trimaran support team, and Rodric Rabbah in particular, for answering numerous questions about Trimaran. This project is supported by the Defense Advanced Research Projects Agency under contract F33615-01-C-1892, NSF CAREER grants CCR-9985109 and CCR-9984336, two IBM University Partnership awards, and a grant from the Intel Research Council.

References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, June 2000.

[2] Arvind and R. S. Nikhil. Executing a program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, 1990.

[3] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 45–54, July 1998.

[4] D. Burger and T. M. Austin. The SimpleScalar tool set version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[5] H. Corporaal. *Transport Triggered Architectures*. PhD thesis, Delft University of Technology, September 1995.

[6] H. Corporaal and H. Mulder. Move: A framework for high-performance processor design. In *Supercomputing-91*, pages 692–701, November 1991.

[7] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, April 1991.

[8] W. J. Dally. Express cubes: Improving the performance of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 40(9):1016–1023, September 1991.

[9] J. Dennis and D. Misunas. A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture*, pages 126–132, January 1975.

[10] J. Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the Tenth Annual International Symposium on Computer Architecture*, pages 140–150, June 1983.

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

[12] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal Q1*, 2001.

[13] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March 1999.

[14] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[15] J. Llosa, M. Valero, J. Fortes, and E. Ayguade. Using sacks to organize register files in VLIW machines. In *CONPAR 94 - VAPP VI*, pages 628–639, September 1994.

[16] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, June 1992.

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

[18] L. Peh and W. J. Dally. Flit-reservation flow control. In *Proceedings of 6th International Symposium on High-Performance Computer Architecture*, pages 73–84, January 2000.

[19] N. Ranganathan and M. Franklin. An empirical study of decentralized ILP execution models. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 272–281, October 1998.

[20] B. Rau. Dynamically scheduled VLIW processors. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 80–90, December 1993.

[21] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 138–148, December 1997.

[22] The national technology roadmap for semiconductors. Semiconductor Industry Association, 1999.

[23] M. Smelyanskiy, G. Tyson, and E. Davidson. Register queues: A new hardware/software approach to efficient software pipelining. In *International Conference on Parallel Architectures and Compilation Techniques (PACT 2000)*, pages 3–12, October 2000.

[24] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 414–425, June 1995.

[25] M. B. Taylor, J. Kim, J. Miller, F. Ghodrat, B. Greenwald, P. Johnson, W. Lee, A. Ma, N. Shnidman, V. Strumpen, D. Wentzlaff, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw processor - a scalable 32-bit fabric for embedded and general purpose computing. In *Proceedings of Hot Chips XIII*, August 2001.

[26] Trimaran : An infrastructure for research in instruction-level parallelism. http://www.trimaran.org.

[27] A. K. Uht, D. Morano, A. Khalafi, M. de Alba, T. Wenisch, M. Ashouei, and D. Kaeli. IPC in the 10's via resource flow computing with Levo. Technical Report 092001-001, Department of Electrical and Computer Engineering, University of Rhode Island, Kingston, RI, September 2001.

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

[29] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarsinghe, and A. Agarwal. Baring it all to software: RAW machines. *IEEE Computer*, 30(9):86–93, September 1997.