Adapting Software Pipelining for Reconfigurable Computing

Timothy J. Callahan and John Wawrzynek
University of California at Berkeley

ABSTRACT

The Garp compiler and architecture have been developed in parallel, in part to help investigate whether features of the architecture help facilitate rapid, automatic compilation utilizing the Garp's rapidly reconfigurable coprocessor. Previously reported work for compiling to Garp has drawn heavily on techniques from software compilation rather than high-level synthesis. That trend continues in this paper, which describes the extension of those techniques to support pipelined execution of loops on the coprocessor. Even though it targets hardware, our approach resembles VLIW software pipelining much more than it resembles hardware synthesis retiming algorithms.

This paper presents a simple, uniform schema for pipelining the hardware execution of a broad class of loops. The loops can have multiple control paths, multiple exits (including exits resulting from hyperblock path exclusion), data-dependent exits, and arbitrary memory accesses. The Garp compiler is fully implemented, and results are presented. A sample benchmark, wavelet image encoding, saw its overall speedup on accelerated loops grow from a speedup of about 2 without pipelined execution to a speedup of about 4 with pipelined execution.

1. INTRODUCTION

Programmable system-on-a-chip components, containing a microcontroller and reconfigurable hardware, promise "the flexibility of software and the performance of hardware". But the reconfigurable hardware is not truly hard since it is programmable, so where does its benefit come from? Its advantage is that it enables a *spatial* model of computing that is traditionally associated with hardware, compared to a microprocessor's *temporal* fetch-and-execute model [5]. Applications often contain a mix of computation, some better suited to spatial computation and some better suited to temporal computation, motivating hybrid architectures that can support both efficiently. Our group designed such an architecture, Garp [7, 4], which features a single-issue microprocessor with a rapidly reconfigurable array attached as a coprocessor for loop acceleration.

Since our research group is investigating reconfigurable hardware primarily in the context of *general purpose* computing, compiler development is an integral part of our architecture research. In contrast, automatic compilation to reconfigurable architectures has been considered less important in the embedded domain since most designers have hardware expertise. However, as embedded applications grow in size, the productivity benefits of automatic compilation also grow. Designers can start with automatic compilation, then focus their efforts on improving a few loops while benefiting from the compiler's speedup on the remainder. Furthermore, with the compiler's support, the designer's required effort is reduced in many cases to simply restructuring the code or embedding compiler directives.

The Garp compiler's challenge is to exploit the reconfigurable hardware's parallelism starting from a sequential software program. We draw heavily on compilation techniques for VLIW architectures, which also exploit operation-level parallelism. In particular, this paper focuses on our adaptation of VLIW software pipelining techniques. This has been an instructional experience for us, as it has in fact highlighted some key differences between spatial and temporal computation using reconfigurable hardware vs. VLIW respectively.

2. THE GARP ARCHITECTURE

Garp's coprocessor is a two-dimensional array of configurable logic blocks (CLBs). Array details dictate that modules run horizontally along a row (Figure 1): fast, flexible carry chains running along each row enable fast sums and comparisons, while horizontal wire channels between rows enable fast shifts between adjacent modules. Vertical buses of various lengths connect modules to each other to form a standard bitslice layout. Each row contains the equivalent of two 32-bit registers across its middle section; one is typically used at a module's output, while the other can be used to buffer data arriving over a long bus, or to receive data arriving from memory.

Memory buses provide the path into and out of the reconfigurable array. Garp's array has four 32-bit data buses and one 32-bit address bus. While the array is idle, the processor can use the memory buses to load configurations or to transfer data between processor registers and array registers. While the array is active, it is master of the memory buses and uses them to access memory—the same memory system as viewed by the microprocessor, including all lev-

CASES'00, November 17-19, 2000, San Jose, California.

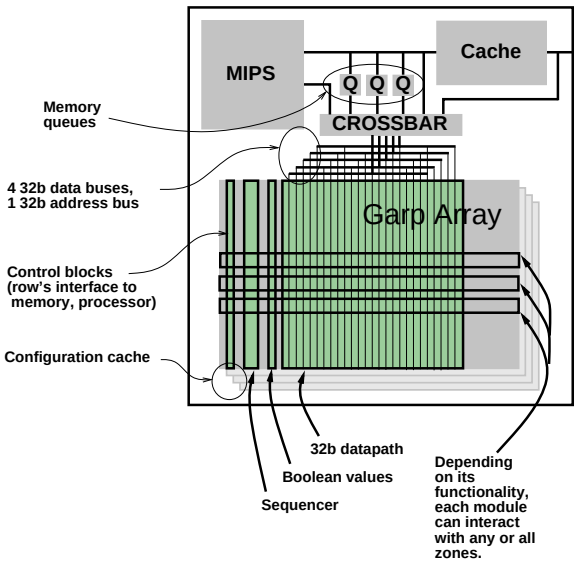

Figure 1: Garp Architecture and Usage Conventions

els of caching. The array initiates an access by sending an address from one row; another row sends or receives the associated data. The Garp architecture also has memory queues for streaming data into and out of the array. When a queue is utilized, the array does not provide the access address, as the starting address and stride are provided by the microprocessor when the queue is initialized.

One of Garp's novel features is the array's fixed clock frequency and simplified timing model. The timing model enumerates exactly which combinations of routing and computation delay in series can be traversed in one clock cycle or less. For example, a module can receive an operand over a short vertical bus and perform a carry chain computation within a single cycle.

3. BASIC COMPILATION APPROACH

The Garp compiler targets loops for acceleration using the reconfigurable array. The compiler utilizes a straightforward approach where the one thread of control hops back and forth between the microprocessor and the array. When a loop utilizing the array is encountered during the execution of a program, the correct configuration is loaded; this takes minimal time if it was the last one used or if it is resident in the configuration cache. Live values are transferred to the array; the array is activated and the microprocessor goes to sleep. When the loop exits, the microprocessor wakes up, determines which exit was taken, retrieves live variables from the array as appropriate for that exit, and continues software execution.

Loops. The Garp-specific part of the compiler begins by recognizing *natural loops* [1] in the control flow graph. These are recognized whether they originate from FOR loops, WHILE loops, or even backwards GOTO statements in the C source code. All are candidates for acceleration. Break and continue statements are allowed. All loops are treated as WHILE loops (having data-dependent exits).

Hyperblock Selection. From each loop we form a hyperblock [13], which contains only the commonly executed paths in the loop. Only the hyperblock is implemented on the array [2]. When an excluded path must be executed, it is executed on the microprocessor, involving a transfer of control back to software; control is returned to the array at the beginning of the next iteration. Excluding paths leads to smaller and faster array configurations, but if an excluded path is executed too often, the extra overhead of switching back and forth between hardware and software will negate any benefits from the exclusion. The exact heuristic for excluding paths differs somewhat from that used for VLIW compilation, but the main idea is the same: to minimize overall execution time. Profiling data is used when available. User comments can optionally be inserted in the code to force the compiler to include or exclude certain paths, or to force all of a loop to execute in software.

As an example, consider a path that is executed one percent of the iterations. If excluded it will incur 30 cycles of overhead each time it is executed, but its exclusion would reduce the critical path by one cycle. This causes extra overhead on average of roughly $0.01 \times 30 = 0.3$ cycles per iteration, but it saves one cycle per iteration, so the exclusion in this case is a net gain.

Paths that contain operations not implementable on the array, for example subroutine calls that cannot be inlined or nested inner loops, also must be excluded. If it is impossible to form a "good" hyperblock—one that will contain most iterations and can be implemented on the array—the entire loop must be rejected from consideration for hardware acceleration.

Loop backedges are considered to be internal to the hyperblock. This allows us to treat all hyperblock exits (branches to a basic block outside of the hyperblock) uniformly, whether they result from branching to an excluded path or from a normal loop exit.

DFG Formation. Next, a dataflow graph (DFG) is constructed from the basic blocks selected for the hyperblock. As the DFG is built, all control dependence is converted to data dependence through the introduction of predicates. Multiplexors controlled by predicates select data values at points corresponding to control flow merge, while each store and exit has a direct predicate input to enable its execution during only those iterations it should be active. Loads can be executed speculatively and so do not require a predicate input. Branches exiting the hyperblock are translated to *Exit* nodes in the DFG.

The DFG contains 3 types of edges:

- *Data edges* make all data dependences explicit—each operation knows exactly which other operation defines each of its operands. The process of resolving dependences is similar to conversion to static single assignment (SSA) form, with the inserted multiplexors superseding SSA's phi functions.
- *Precedence edges* are added between operations that must be ordered but do not have an explicit data dependence, in particular, between pairs of memory accesses that may be aliased. Array and pointer alias analysis is used to eliminate precedence edges between memory accesses whenever possible.
- *Liveness edges* to Exit nodes are used to indicate which

values are live at each exit. They also force an ordering: the Exit cannot be taken before the live value has been computed.

All edge types are labeled with their *distance*: how many iteration boundaries they cross. Thus an edge within an iteration has distance 0, while a loop-carried edge has distance 1 or greater. This allows uniform treatment of intra- and inter-iteration edges.

DFG Optimization. Often direct translation of the C language's short circuiting branch semantics leads to inefficient multiplexor constructs. An optimization pass transforms layers of multiplexors to a single multiplexor controlled by a Boolean expression of comparison results. Other optimizations include strength reduction, redundant load elimination (including reuse of array elements by subsequent iterations), and recognition of memory accesses that can utilize the memory queues. The latter two memory optimizations work with both pointer and array accesses.

Datapath Synthesis and Execution. The optimized DFG is then implemented as a direct, fully spatial network of modules on the array—every operation gets its own hardware that never has to be time-multiplexed to execute other operations. Our fully spatial approach allows groups of adjacent DFG nodes to be merged into optimized modules [3], reducing the area and critical path delay. We will also see that the spatial approach greatly simplifies pipelining, described in the next section.

The grouping of DFG nodes into optimized modules is performed by a novel algorithm that resembles instruction selection. Each resulting module can be considered to be a complex instruction that ultimately is statically bound to a row in the array. A grammar is used to parse the DFG, recognizing patterns in the DFG known to correspond to realizable modules. This grammar is ambiguous; a cost function is used to choose the parses that result in the smallest and fastest datapaths. The "instruction set" — the set of known modules from which to choose — is enormous due to the flexibility of the Garp array, conservatively on the order of tens of thousands. To keep the size of the grammar reasonable, common subpatterns are factored out. Ultimately each pattern is fed to a generator that can produce the equivalent module configuration. The grammar/generator combination has embedded in it human expertise for using the array effectively at the module level, while the parsing algorithm uses the computer's power to find the best combination of modules to globally optimize the datapath.

The algorithm simultaneously considers relative placment of the modules, allowing interconnect latencies to be considered. It also allows special optimizations between adjacent modules. Placement constraints such as "adjacent" or "close" are added to inter-module edges; during subsequent global placement of modules, these constraints must be respected. Care is taken so that conflicting constraints never occur.

Most modules have a latency of one cycle for each data input, considering the delay from the output register of the source module, over interconnect, through the module's computation (lookup table and optionally carry chain), to the module's output register. In some cases, though, an input's path may require multiple cycles, in which case registers are used to break it up in to single-cycle segments. Thus every module is single-cycle pipelined by construction.

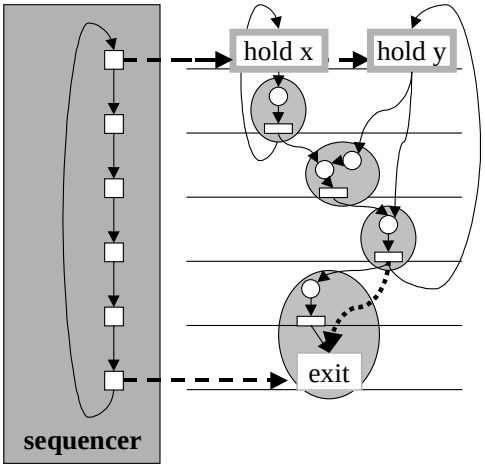

Figure 2: Non-pipelined array execution. This figure portrays the scheduling, not physical placement. Solid lines are data edges. Dashed lines are triggers from the sequencer. Dotted lines are liveness edges.

The datapath contains a Hold module for each loop-carried variable. Before execution begins, each Hold module is loaded with its value at loop entry by the microprocessor. Once array execution begins, modules scheduled at the first cycle have their valid inputs and produce a valid result a cycle later. Then modules scheduled in the next cycle have valid inputs and will subsequently produce their valid output for that iteration, and so on, until all modules have computed their valid outputs or performed their action. The iteration is completed after SL cycles, where SL is the schedule length, the cycle of the last-computed module. At the end of the first and each subsequent iteration, each Hold module latches the new value for its variable for use in the next iteration. Iteration continues indefinitely until an Exit node receives a TRUE input during the cycle it is scheduled.

Some modules such as those involved in Exits and memory accesses must be triggered only during the appropriate cycle of each iteration, since execution before their inputs are valid would have bad consequences. Also, Hold modules need a trigger at the end of each iteration so that they load the loop-carried values for use during the next iteration. These triggers are provided by a simple *sequencer* that is synthesized within the configuration. Its duty is to keep track of the current cycle within the iteration and provide the trigger to modules as needed. The implementation of the sequencer is trivial since every iteration has the same fixed schedule. It is simply a Boolean shift register of SL stages, hooked back on itself, passing around a single '1' bit.

In contrast, other modules such as adders need no trigger as they can execute constantly with no ill side effects. However, it is useful to think of these non-triggered modules as also being "scheduled" for a specific cycle: the cycle of the iteration that their output first becomes valid.

4. PIPELINED APPROACH

In the case of Garp, pipelining is accomplished mainly through rescheduling the execution of the modules on the array, which is one of the last compilation steps. However, to get the most out of pipelining, two changes are needed during earlier phases of compilation. Both result from the fact that pipelined performance is usually governed by the longest feedback loop in the computation (the *critical cycle*) instead of the longest path through one iteration (the *critical path*).

- When considering a path to include or reject from the loop, the hyperblock selection algorithm now evaluates the impact on the critical cycle rather than the critical path.
- The module mapping algorithm, when packing DFG nodes into feasible modules, similarly adjusts its goal to minimizing the critical cycle instead of minimizing the critical path.

After module mapping, each module is considered as an atomic unit. Its scheduling in time as well as its placement on the array are still flexible. Recall that modules are completely pipelined, with a register inserted every cycle of delay on internal paths.

4.1 Modulo Scheduling

Because multiple iterations will be active simultaneously on the network of modules, care must be taken so that non-queue memory accesses from different iterations do not conflict. We utilize a scheduling algorithm directly based on Rau's iterative modulo scheduling (IMS) [17].

Modulo scheduling is a framework for scheduling a single iteration of a loop in a way that resolves resource conflicts among consecutive overlapping iterations [9]. Successive iterations have identical schedules, shifted by a number of cycles called the initiation interval (II). Resource conflicts between overlapping iterations are captured by folding the schedule of the single iteration back on itself, modulo II cycles.

In brief, we implement IMS as follows. A lower bound on II is found by using the greater of the recurrence-constrained minimum II and the resource-constrained minimum II. IMS attempts to find a valid schedule for this minimum II; if it fails, it tries again with an II incremented by one, and repeats until a valid schedule can be found. A worklist algorithm is used to search for a valid schedule for each given II. The worklist contains unscheduled modules, and initially contains all modules. As each module is removed from the worklist, it is scheduled for the earliest cycle that does not conflict with any other scheduled module in terms of precedence or modulo resource constraints. If no such cycle exists, the module is still scheduled for some cycle, but other conflicting scheduled modules are descheduled and returned to the worklist. The algorithm terminates successfully if the worklist becomes empty (all modules have been scheduled). It terminates with failure for that value of II if *Budget* scheduling steps have been performed and there are still unscheduled modules. *Budget* is set to a small multiple of the number of modules.

After modulo scheduling, the compiler removes extra slack on edges between different strongly-connected components (SCCs) where possible. If an SCC has no global resource usage, it can be shifted arbitrarily while still leaving a valid schedule. If it does use global resources, it can be shifted only by multiples of II cycles.

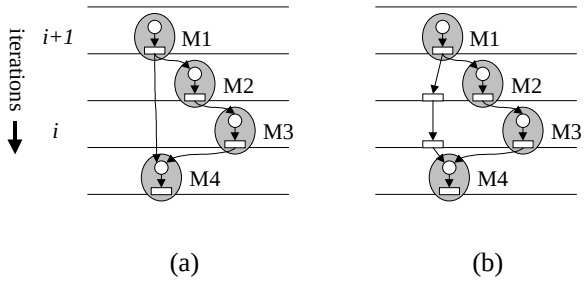

Figure 3: Register insertion for pipelined execution with II=2 (see text). (a) M4 will incorrectly see operands arriving from different iterations. (b) After register insertion, operands arrive at M4 correctly.

4.2 Register Insertion

The overlap of iterations causes a problem as shown in Figure 3(a). Consider a schedule with II=2. Module M1 will produce its output for a certain iteration i at cycle 1 of that iteration. That value is correctly consumed immediately by module M2. However, a problem develops with module M4 at cycle 4 of iteration i. At that point, module M1 has already computed its output for the following iteration $i+1$, but M4 is expecting to receive M1's output for iteration i (although at the same instant, M2 *is* expecting M1's output for iteration $i+1$). This problem directly corresponds to the overlapping lifetime problem encountered in VLIW compilation.

The solution is to insert registers on edges such as that from M1 to M4 (Figure 3(b)). This ensures that partial results from the same iteration arrive at a module's inputs with the correct timing.

Each data and liveness edge is examined to see if it requires the insertion of registers. The insertion of registers on liveness edges ensures that the correct version of a variable is available when an exit is taken, and is somewhat analogous to the modulo variable expansion to remove live-out anti-dependences described in [10].

The number of registers that must be added to an edge is equal to the slack on the edge after scheduling; this is why the compiler attempted to remove excess slack between SCCs after modulo scheduling. Edges in critical cycles will not need any additional registers. If a module has fanout, the same register chain is shared by the different consuming modules.

With the addition of these registers, the progress of an iteration—its partial results and external actions such as memory accesses—is confined to exactly one level in the schedule. This ensures that multiple iterations can utilize the hardware without interfering with each other.

Since timing on all paths is now exactly synchronized, there is no need for the Hold modules inserted on each loop carried edge. Just as data is shifted synchronously forward, the passing of loop-carried variable values is timed to exactly meet the values of the correct subsequent iteration.

After registers are inserted, placement is again performed on the datapath so that the new registers intermingle with

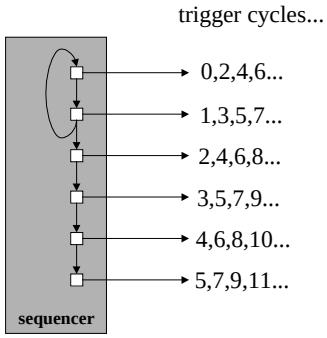

Figure 4: Sequencer for pipelined execution with II=2.

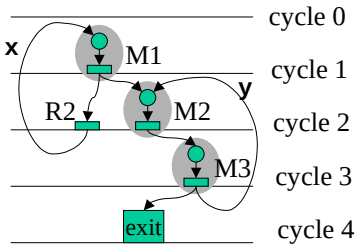

Figure 5: Loop-carried variable initialization during prologue. II=2. The microprocessor writes the value of variable 'x' to register R2 at cycle 0 for use by M1 during the first iteration. Similarly, the value of variable 'y' is written to the output register of M3 at cycle 1 for use by M2.

the original modules. When one or more registers are inserted along an edge that has placement constraints ("close" or "adjacent"), those constraints are transferred to the edge between the last inserted register and the original edge destination. No placement constraints are needed on edges between two registers or between the orignal source and the first register.

Adding registers can be inefficient when II is moderate to large. Considering a long chain of inserted registers, only one out of every II registers actually contains a useful value. The other II-1 out of II registers contain "don't care" values. When II is large it is more efficient to instead use Hold modules (in the case of Garp, it is beneficial to use Hold modules instead of simple registers when II > 2, since each row can implement either one Hold module or two simple registers). With Hold modules, the values are shifted every II cycles instead of every cycle.

4.3 Sequencer

The sequencer's implementation was trivial with non-pipelined execution, and is still quite simple with pipelined execution. While the sequencer's shift register length is still equal to the schedule length SL [1], the feedback loop is different, so that subsequent iterations are started *before* the current is done. Specifically, the feedback edge originates from the II'th tap rather than the SL'th tap. Thus, a sequencer output from tap t is activated at cycles t, $t + II$, $t + 2 \times II$, etc. (Figure 4).

4.4 Prologue

Pipelined computation, whether in software or hardware, characteristically has a *prologue*, the period from the start of computation to the time the first iteration completes, at which point the pipeline achieves a steady state. Specifically, during the prologue, latter portions of the pipeline should not be active, since the first iteration has not yet reached them. With VLIW processors, a number of schemas for handling this have been put forward [15]. One category emits the prologue as separate code, with inactive pipeline stages simply eliminated. Another solution, termed "kernel-only", uses the same instructions for both the prologue and the steady-state execution, but uses *stage predicates* to suppress inactive stages during the prologue. The sequencing technique utilized by the Garp compiler more closely resembles the latter, since each iteration maintains the same schedule, even those started during the prologue. In some sense, the sequencer combines the functions of program counter (activating "instructions" 0...II-1) with stage predicates.

[1] The schedule length SL may have been extended compared to the basic (nonpipelined) case in order to resolve modulo resource conflicts.

4.5 Initial Values

Loop-constant inputs are handled the same as in the basic schema. Before array execution begins, they are transferred into their appropriate registers where they remain constant throughout all iterations. Similarly, queues are initialized by the main processor same as with non-pipelined execution.

Initial values of loop-carried variables are handled differently, however, because Hold modules will not necessarily exist on every loop-carried data edge. We could build special circuitry into the datapath to handle variable initialization, but this would add complexity and possibly slow down the steady-state pipelined execution rate. So instead, we utilize the Garp architecture's close connection between the MIPS processor and reconfigurable coprocessor, and transfer the complexity into software (the software does not *perform* the prologue, it simply helps provide the initial values during the prologue).

Each module input that is fed by a loop-carried data edge must be supplied an initial value during the first iteration (Figure 5). For each such input, the main processor writes into the supplying register the correct initial value at precisely the correct cycle for use by the first iteration. Since there maybe be number of such initial values required at different cycles of the prologue, the processor will typically alternate between writing one or more initial values, then activating the array for a small number of cycles, then writing another initial value, and so on until all initial values for use by the first iteration have been supplied. After the last initial value has been supplied, the processor activates the array for an indefinite period, that is, until the array halts itself.

If a loop-carried variable is used by multiple modules, usually only one write of the initial value is required. If the consuming inputs are all required the same cycle, they would be supplied by the same register (whether the source module's output, or a delayed version thereof). If the consuming inputs are needed different cycles, only the earliest-used register in the chain need be initialized; this initial value will then propagate down the rest of the chain supplying later-needed inputs. One way of thinking about this initial value

	SL		II		Area (rows)		Speedup over MIPS	
Loop	Basic	Piped	Basic	Piped (minRecII,minResII)	Basic	Piped	Basic	Piped
entropy_encode_544	3	3	3	1 (1,0)	4	3	3.32	9.88
entropy_encode_557	8	9	8	5 (5,2)	20	27	1.25	1.56
forward_wavelet_647	13	12	13	2 (1,2)	16	26	1.44	5.09
forward_wavelet_674	3	3	3	1 (1,1)	9	9	3.75	6.55
forward_wavelet_696	12	13	12	2 (1,2)	16	26	1.23	2.05
forward_wavelet_711	3	5	3	2 (1,2)	11	13	4.12	4.94
init_image_354	3	3	3	1 (1,0)	5	4	4.24	12.62
RLE_encode_509	6	6	6	1 (1,1)	10	12	2.10	4.58
block_quantize_411	3	4	3	2 (2,1)	10	11	4.55	6.08
Overall (kernels only)							2.09	4.09

Table 1: Statistics from benchmark for schedule length, achieved initiation interval, and area for basic vs. pipelined hardware execution. (minRecII/minResII) give the recurrence- and resource-constrained minimum II, respectively. Speedup includes reconfiguration and data transfer overhead as well as cache misses.

of the variable is that it appears as the output of the module from a fictitious iteration –1.

4.6 Epilogue

Software pipelining of counted loops traditionally involves the generation of an epilogue: special code to finish off the iterations that are in progress at the point when the last iteration is started.

The Garp compiler takes a simpler approach that does not require an epilogue: it simply allows the last iteration to complete using the array, then discards work that has started speculatively on subsequent iterations. This approach is possible because of the Garp architecture's support for speculative execution, and in particular, speculative loads (a subsequent iteration might execute a load with an invalid address). Besides being simpler, this approach also has the benefit that it works for loops with data-dependent exits, allowing a uniform pipelining schema for all loops.

It could be argued that a special epilogue should be generated for counted loops, since it can be optimized to avoid doing unnecessary work. However, in the case of Garp, in practically all cases an optimized epilogue in software is still much slower than simply completing the last iteration in hardware, even if 'extra' work is done in the latter case.

The copying of registers from the array to the main processor register file at loop exit resembles the "live out copying" required by many VLIW software pipelining schemas [15, 10]. They are similar in that the source of each live variable value, and even *which* variables are live, depends on which exit is taken. However, with Garp this copying must be performed even when reconfigurable hardware execution is not pipelined.

5. RESULTS

To quantify the effects of pipelining, we compiled and simulated execution of a representative embedded application, wavelet image compression [8]. This benchmark stresses reconfigurability by splitting execution time among several loops, which together capture 87 percent of the original software execution time. The results shown are from processing a 256×256 pixel image.

The Garp compiler is fully implemented. These results were generated by completely automatic compilation of the C source code, with no embedded hints or directives.

5.1 Pipelined vs. Not Pipelined

For each kernel, Table 1 provides a comparison of basic vs. pipelined execution on the reconfigurable array. The first columns compare the schedule length and initiation interval for basic vs. pipelined. In most cases pipelining is successful at significantly reducing the initiation interval. In all cases the minimum possible II is met. The resource-constrained minimum II for a loop is equal to the number of non-queue memory accesses in the loop; sometimes this is the greater constraint, and in other cases a recurrence is the greater constraint. The pipelined schedule length is in some cases slightly longer (due to conflict resolution in modulo scheduling) and in other cases is slightly shorter (due to the elimination of a Hold module for a loop-carried variable).

In most cases the pipelined configuration is larger due to the addition of registers, 62.5% larger in the worst case. However, in many cases the increase is negligible, and in two cases the pipelined configuration is actually smaller, again due to the elimination of Hold modules.

The final set of columns present relative speedup of non-pipelined and pipelined execution over just the MIPS. These values are derived from our cycle-accurate Garp simulator, which accurately models first and second level cache hit and miss latencies. The speedup values given include all overhead costs, including reconfiguration and data transfer.

In many cases the relative performance of the pipelined version falls short of what would be predicted by simply considering the ratios of the initiation intervals. That is because overhead costs associated with using the array were significant, and do *not* scale down with the II. Instead, they stay constant or even increase: possibly larger pipelined configurations can incur more reconfiguration cost, and the pipeline prologue, usually involving starting and stopping the array, also adds more overhead. Finally, cache misses will dilute speedups. Still, substantial benefit overall is gained by pipelining the execution. Also, this benchmark had many short-running loops; with longer running loops, the overhead costs would be less significant.

This benchmark also reinforces how important it is for the compiler to utilize Garp's memory queues whenever possible. Queue utilization is important for pipelining since it reduces contention for the bus. Each of the forward_wavelet loops has four memory accesses per cycle. If none used the queues, the achievable throughput would be just one itera-

	II		ILP		
Loop	Garp	VLIW	Garp	VLIW	*
entropy_encode_544	1	1	5	5	
entropy_encode_557	5	5	3.4	3.4	
forward_wavelet_647	2	9	10	2.2	a
forward_wavelet_674	1	4	13	3.3	a
forward_wavelet_696	2	9	10	2.2	a
forward_wavelet_711	2	4	7	3.5	a
init_image_354	1	3	8	2.7	a
RLE_encode_509	1	2.25	11	4.9	b
block_quantize_411	2	2.5	5.5	4.4	c

Table 2: Comparison of compilation for Garp vs. 6-wide VLIW architecture. *Notes: (a) Weak dependence analysis affected Trimaran VLIW results. (b) Could not be software pipelined by Trimaran because it was a WHILE loop; unrolled by a factor of four instead. (c) Could not be software pipelined by Trimaran because hyperblock formation excluded paths; unrolled by a factor of four instead.

tion every four cycles, since each access would need a cycle to use the address bus. But since two or three of the accesses can be converted to queue accesses in each case, an II of two or one is achieved.

5.2 Comparison with VLIW

For comparison we compiled the same benchmark using the Trimaran 2.0 VLIW compiler [19]. The results targeting the default 6-wide VLIW architecture are shown in Table 2. It should be noted that the Trimaran compiler is also a prototype compiler and is also undergoing constant improvement, so these results should be considered to be a snapshot of work in progress.

For many of the loops, the achieved ILP was limited by Trimaran's weak dependence analysis (which will be improved in the next release). However, even with better dependence analysis, it would not be able to achieve ILP greater than 6 becaues of the instruction issue limit; Garp exceeds this amount in each of those loops. This also illustrates how important good dependence analysis is for achieving high ILP—and how it's even more important with Garp because of the higher peak ILP that can be exploited.

Two other loops could not be pipelined by Trimaran because they were not counted FOR loops. However, Trimaran was able to exploit inter-loop ILP by instead unrolling the loops. Unrolling is not a good alternative for Garp in most cases because it increases reconfigurable resource demands, so it is important that the Garp compiler is able to utilize pipelining for these types of loops.

6. DISCUSSION

From the compiler's point of view, the main difference compared to VLIW modulo scheduling is that our case has very few and very simple resource conflicts. We do not face per-cycle instruction issue limits or complicated instruction bundling rules. Each module *is* its own resources and so does not interfere with the execution of other modules. There is not even a limit on the number of exits (branches out of the hyperblock) evaluated per cycle. The single global shared resource that must be allocated among the different modules is the address bus used for non-queue memory accesses.

It is not surprising that pipelining is more straightforward with Garp's reconfigurable hardware, since pipelining is essentially a spatial concept. In contrast to a hardware pipeline in which all operators can execute simultaneously, the VLIW processor can perform only a small number (and then only certain combinations) each cycle. While this is obviously a limitation, we realized that for a VLIW processor, *moving* the data through the virtual pipeline is even more of a challenge than processing the data. The register chains added to the module pipelines in Garp represent enormous data transfer bandwidth, possible only because each register has its own read and write port and thus all can be accessed simultaneously. A direct implementation of such a register chain in software on a VLIW architecture using register-to-register move instructions would be intolerably slow because of instruction and register file bandwidth limits. Rotating register files [16] are a clever circular buffer implementation of a shift register that eliminates the need for register-to-register moves. Modulo variable expansion [9] is a software version of this same circular buffer trick.

7. RELATED WORK

All known related work in pipelined compilation of loops for reconfigurable hardware can handle only constant-stride array-based memory accesses. This is mainly due to their target architectures, which do not facilitate low-latency fetches of arbitrary memory locations. They assume all memory access occurs via programmable external DMA engines similar to Garp's memory queues, or to local copies of arrays.

Other projects as currently implemented are limited to pipelining counted FOR loops without breaks. Furthermore, no known related work has the capability to exclude rarely-executed portions of the loop. In their cases this might not be practical because of the assumed high overhead for switching from hardware to software execution and back.

Weinhardt has investigated pipelined datapath generation using vector compilation techniques [18]. An important part of his compilation flow is the reduction of memory traffic by reusing data for index-shifted accesses to the same array. His synthesis approach more closely resembles retiming algorithms [11], in that it starts by considering the acyclic portion of the DFG to be purely combinational, and then moves in registers to minimize clock cycle time and/or added area. Similar work in progress is described in [14].

The NAPA-C compiler [6] is similar to the Garp compiler in that it utilizes iterative modulo scheduling to resolve memory scheduling conflicts. However, its target's array clock frequency is not fixed but variable, determined by the latency of the longest operator, so it is not clear whether memory ports are always utilized at their full capacity. The NAPA-C compiler relies on user annotations to partition data and code between the main processor and the reconfigurable coprocessor.

8. CONCLUSION

We have presented our schema for overlapping the execution of iterations on Garp's reconfigurable array to increase throughput. These techniques were directly adapted from VLIW compilation, and in many cases simplified in the process. We consider it to be a positive result that our compiler can use simpler techniques to achieve greater parallelism in

many loops, targeting what is in fact a *simpler* architecture than most contemporary VLIW architectures.

The Garp compiler specifically exploits many of the Garp chip's novel features, since one of its goals was to evaluate their usefulness. It should not be concluded, however, that the Garp compiler's techniques are not retargetable to differing architectures (although we do anticipate that future hybrid reconfigurable architectures will incorporate some of Garp's features). A related project undertaken at Synopsys [12] has extended the Garp compiler in many ways, producing a version that targets not only Garp but also a hybrid platform using a Xilinx Virtex chip as its reconfigurable coprocessor. Furthermore, almost none of the Garp compiler's flow assumes bit-level reconfigurability, and thus it could be easily retargeted to coarser-grain reconfigurable fabrics.

Future work will focus mainly on continuing evaluation of the Garp architecture and compiler over a broad range of benchmarks.

9. REFERENCES

[1] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers, Principles, Techniques, and Tools* . Addison-Wesley Pub. Co., 1986.

[2] Callahan, T., and Wawrzynek, J. Instruction-Level Parallelism for Reconfigurable Computing. In *Field-Programmable Logic and Applications. 8th International Workshop, FPL'98. Proceedings* (1998), Springer-Verlag, pp. 248–57.

[3] Callahan, T. J., Chong, P., DeHon, A., and Wawrzynek, J. Rapid Module Mapping and Placement for FPGAs. In *Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey CA USA, 1998), ACM, pp. 123–132.

[4] Callahan, T. J., Hauser, J. R., and Wawrzynek, J. The Garp Architecture and C Compiler. *IEEE Computer 33*, 4 (Apr. 2000), 62–69.

[5] DeHon, A., and Wawrzynek, J. Reconfigurable Computing: What, Why, and Implications for Design Automation. In *Proceedings 1999 Design Automation Conference* (1999), IEEE, pp. 610–15.

[6] Gokhale, M., Stone, J., and Gomersall, E. Co-synthesis to a hybrid RISC/FPGA architecture. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology 24*, 2 (Mar 2000), 165–80.

[7] Hauser, J., and Wawrzynek, J. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines* (Napa, CA, Apr. 1997), K. L. Pocek and J. M. Arnold, Eds.

[8] Kumar, S., Pires, L., Pandalai, D., Vokta, M., Golusky, J., Wadi, S., and Spaanenburg, H. Benchmarking Technology for Configurable Computing System. In *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines* (1998), IEEE Comput. Soc, pp. 273–4.

[9] Lam, M. S. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the ACM SIGPLAN 1988 Converence on Programming Language Design and Implementation* (June 1988), pp. 318–28.

[10] Lavery, D., and Hwu, W. Modulo Scheduling of Loops in Control-Intensive Non-Numeric Programs. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture.* (1996), IEEE Comput. Soc. Press, pp. 126–37.

[11] Leiserson, C. E., and Saxe, J. B. Optimizing Synchronous Systems. *Journal of VLSI and Computer Systems* (1983), 41–67.

[12] Li, Y., Callahan, T., Darnell, E., Harr, R. E., Kurkure, U., and Stockwood, J. Hardware-Software Co-Design of Embedded Reconfigurable Architectures. In *Proc. 37th ACM/IEEE Design Automation Conference DAC 2000* (June 2000), IEEE Computer Society Press.

[13] Mahlke, S., Lin, D., Chen, W., Hank, R., and Bringmann, R. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture* (Dec. 1992), pp. 45–54.

[14] Moisset, P., Park, J., and Diniz, P. Very High-Level Synthesis of Datapath and Control Structures for Reconfigurable Logic Devices. In *The Second International Workshop on Compiler and Architecture Support for Embedded Systems (CASES'99)* (Oct. 1999).

[15] Rau, B., Schlansker, M., and Tirumalai, P. Code Generation Schema for Modulo Scheduled Loops. In *Proc. 25th Annual International Symposium on Microarchitecture* (Dec. 1992), pp. 158–69.

[16] Rau, B., Yen, D., Yen, W., and Towle, R. The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-offs. *Computer 22*, 1 (Jan 1989), 12–35.

[17] Rau, B. R. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture.* (1994), ACM, pp. 63–74.

[18] Weinhardt, M., and Luk, W. Pipeline Vectorization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (to appear).

[19] See http://www.trimaran.org/.