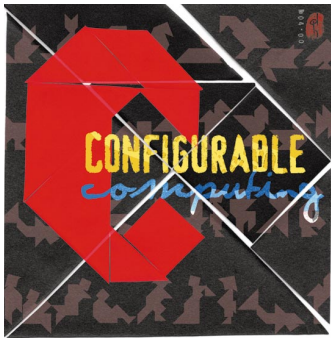
The Garp Architecture and C Compiler



Garp's on-chip, reconfigurable coprocessor was tailored specifically for accelerating loops of general-purpose software applications. Its novel features inspired a unique approach to automatic compilation from C.

Timothy J. Callahan
John R. Hauser
John Wawrzynek
University of California, Berkeley

Various projects and products have been built using off-the-shelf field-programmable gate arrays (FPGAs) as compute accelerators for specific tasks. Such systems typically connect one or more FPGAs to the host computer via an I/O bus. Some have shown remarkable speedups, albeit limited to specific application domains.

Many factors limit the general usefulness of such systems. Long reconfiguration times prevent acceleration of applications that spread their time over many different tasks. Low-bandwidth paths for data transfer limit the usefulness of such systems to tasks that have a high compute-to-memory-bandwidth ratio. In addition, standard FPGA tools require hardware design expertise beyond the knowledge of most programmers.

To address the bandwidth problems, some developers have proposed integrating specially designed rapidly reconfigurable hardware more closely with the processor. To help investigate this idea we designed our own architecture in detail, called Garp,[1] and experimented with running applications on it. We are also investigating whether Garp's design enables automatic, fast, effective compilation across a broad range of applications. Our results so far have been promising.

GARP OVERVIEW

Garp combines a single-issue MIPS processor core with reconfigurable hardware to be used as an accelerator. We designed both the reconfigurable hardware and the interfaces among the system components, tailoring them for general-purpose computing. Garp is designed to fit into an ordinary processing environment that includes structured programs, subroutine libraries, context switches, virtual memory, and multiple users.

The Garp chip does not exist as real silicon. We have, however, completed critical parts of the integrated circuit layout and performed circuit simulation to give us good estimates of Garp's clock speed, power consumption, and silicon area for a sample implementation.

We designed Garp with the intent that its reconfigurable hardware would accelerate loops of general-purpose programs. This goal led to the following decisions about Garp:

- We decided that a few cycles of overhead for transferring data between processor registers and the reconfigurable hardware would be acceptable, since this overhead would occur only at the entrance and exit of loops.
- The reconfigurable hardware needed its own direct path to the processor's memory system, since most nontrivial loops operate over memory data structures. Relying on the main processor to shuffle data between the reconfigurable hardware and memory would be unacceptable; the processor would act as a bandwidth bottleneck and also add cycles of latency to every access.
- The reconfigurable hardware needed to be *rapidly* reconfigurable, since general-purpose applications tend to have many short-running loops.

Garp's reconfigurable hardware attaches to the main MIPS processor as a coprocessor. Explicit processor move instructions transfer data between the two parts. Additional instructions give the main processor complete control over the loading and execution of the coprocessors' configurations. We call Garp's coprocessor "the reconfigurable array" or simply "the array."

0018-9162/00/$10.00 © 2000 IEEE

The wide path between the array and memory is useful not only for data transfer, but also for reducing configuration load times. Further, Garp's dense configuration encoding results in smaller configurations that load quickly. For example, a medium-sized configuration of 480 configurable logic blocks (CLBs) occupies less than 4 Kbytes, which can be read from main memory in only a couple thousand processor cycles or, from the secondary cache, in only a few hundred cycles. To reduce configuration times further, the array has embedded in it a *configuration cache*, which holds recently displaced configurations for rapid reloading. Reloading an entire configuration from the configuration cache requires approximately five cycles. Our sample implementation's configuration cache can hold four full-sized configurations or a larger number of smaller configurations.

Once a configuration loads and starts executing, the array can continue executing independently until it signals it is done. The processor can halt and resume array execution at any time. Configurations can be loaded only when the array is idle. The processor can also examine or change any data in the array while the array is idle. Attempting to load a configuration or access array data while the array is active causes the processor to stall on an interlock until the current array computation completes. This interlock can be interrupted. Array execution continues, and the stalled instruction reexecutes when the interrupt returns.

Array use typically involves four steps each time the program reaches an accelerated loop or *kernel*:

1. Load a configuration. If the configuration is already in the configuration cache, this step takes minimal time.
2. Copy any initial register data to the array with coprocessor move instructions.
3. Start array execution, then issue a wait instruction that interlocks while the array is active.
4. At kernel completion, copy result live registers back from the array.

Steps 1, 2, and 4 represent the overhead cost for using the array. For a net performance benefit, the speedup gained by array execution must outweigh this overhead.

THE ARRAY AS A RECONFIGURABLE DATA PATH

Like an FPGA, the Garp array is a two-dimensional array of CLBs interconnected by programmable wiring. Like the processor, the array has a fixed global clock synchronizing all array operations. Unlike most uses of FPGAs, the speed of this clock remains constant for an implementation and cannot be adjusted by an array configuration.

It is natural, although not required, that 32-bit integer data paths in the array be oriented so that operations such as addition span the central CLBs of each row and are stacked, connected to each other by vertical buses. The CLBs each contain individual 1-bit registers; the collection of these across the datapath portion of a row is often treated as a composite 32-bit *array register*. The extra CLBs on each side of the data-path area are often useful for implementing controllers and computing Boolean data, as shown in Figure 1.

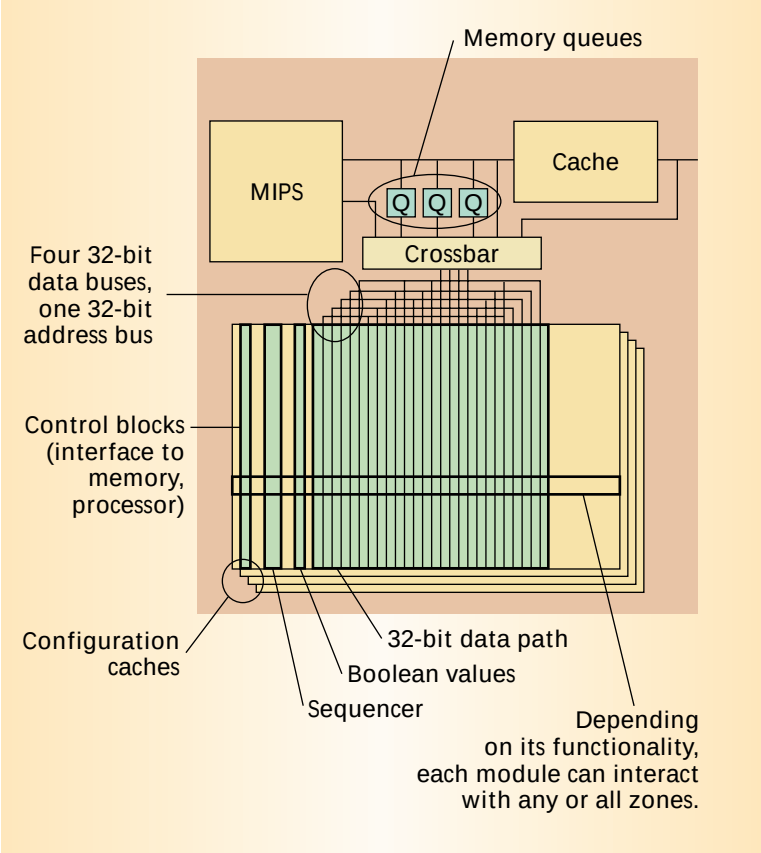


Figure 1. The Garp array, which functions as a reconfigurable data path, consists of a two-dimensional array of configurable logic blocks interconnected by programmable wiring. Memory buses provide a high-bandwidth reconfiguration and data transfer path between the array and memory.

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.

During execution, the reconfigurable array has access to the same memory system as the main processor, including all caches. To perform a random memory write, one row initiates the write and supplies the address, and another row provides the data. Random reads work similarly: Once a row has initiated the read and supplied the address, the data loads into

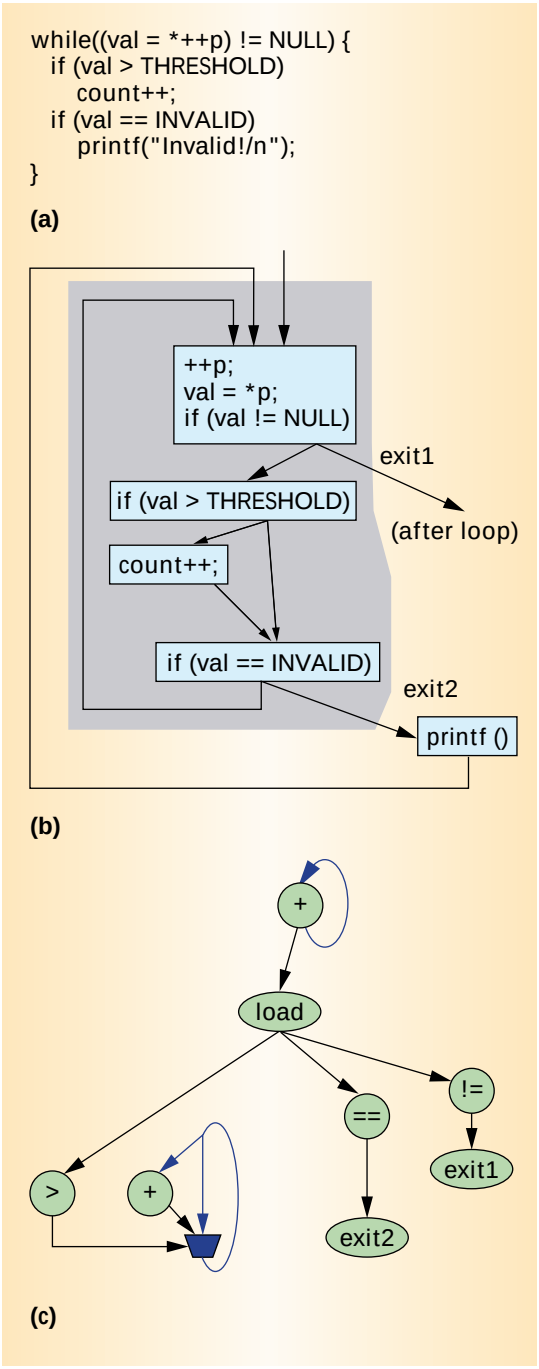


Figure 2. Garp compiler's inner workings. The compiler takes (a) the original source code, processes it into a (b) control flow graph, then for each kernel selects basic blocks included in the hyperblock (gray region). Only computation in the hyperblock is performed in the array. Next, the compiler uses predication to bring together operations from the selected basic blocks into a (c) single large dataflow graph. Comparisons that previously controlled conditional branches now control multiplexors or loop exit nodes. Loop-carried (feedback) edges are shown in blue.

another row after a number of read latency cycles explicitly specified by the configuration. If the memory system cannot respond that quickly, the array stalls automatically, just as a regular processor instruction would interlock on a delayed load. At most, one row can initiate a random access each cycle since there is just one address bus. But the array can overlap accesses, initiating a new one every cycle.

Control blocks reside in the array's leftmost column, one per row, providing the row's control interface to the memory or processor. For example, depending on how a control block is configured, asserting one of its inputs might initiate a memory access, sending 32 bits of data from that row as the address. With an alternative configuration, asserting its input will halt array execution, thus acting as a loop exit.

Along each row of CLBs, built-in carry chain hardware supports efficient additions, subtractions, and comparisons. Horizontal wiring channels between adjacent rows support shifts. These features together make multiplication and division by small constants fairly efficient as well. For example, multiplying a 32-bit variable by any 8-bit constant requires at most two rows and two cycles latency. Because of the flexibility of the CLBs, a single row can often implement a compound group of simple operations. For example, the C integer expressions `(a<<10)|(b&c)` and `(a-2*b+c)` can each be implemented in one array row with a latency of one cycle. We use the term *module* to describe such an implementation of one or more operations.

AUTOMATIC COMPILATION

Our compiler's input is standard ANSI C; the programmer is not required to insert any hints or directives in the source code. Therefore we can use the SUIF C compiler[2] for the front-end phase of compilation—parsing and standard optimizations—with no modification. The compiler breaks up the program into basic blocks, which are instruction sequences with no branches into or out of the middle. At the end of each basic block is a branch that controls which block executes next. These branches connect all the basic blocks of a subroutine into one control flow graph, as shown in Figure 2's section b.

Challenges

The first Garp-specific compilation task identifies the kernels that should be accelerated using the Garp array. One obvious approach is to put every loop that is small enough onto the array, and to execute everything else on the main processor.

Unfortunately this "whole loop" approach does not lead to very good results when compiling typical C source code. Many loops in C programs are large because they include code for exceptional cases that rarely if ever occur. Such loops often will not fit on the array or, if they do, run slowly because of the longer interconnects needed with larger circuits. Also, many loops whether small or large contain operations that cannot be directly implemented on the array, such as `printf` calls.

The C language's inherently sequential nature presents another challenge. Typically, little parallelism exists within each basic block—a drawback considering that to fully exploit the reconfigurable array we must find—and execute in parallel—as many independent operations as possible.

Thus, we face two challenges: excess code in loop bodies, and sequential code from which we must extract instruction-level parallelism (ILP). Fortunately,

Comparing Garp to Other Architectures

The Garp architecture offers several advantages and a few disadvantages when compared to other architectures.

VLIW

As we use it, the Garp array resembles a very-long-instruction-word (VLIW) processor in that the compiler is responsible for scheduling parallel operations. However, Garp does not have VLIW's per-cycle limits on instruction issue, functional unit availability, or register-file bandwidth. Instead, it must limit the size of the entire kernel to be accelerated, a constraint not encountered when compiling for VLIW machines. Garp's array allows the merging of multiple dependent operations into a single module, reducing the critical path. Pipelining on Garp is actually more straightforward than software pipelining on VLIW machines: The overlapping iterations on Garp don't compete for function units, making the scheduling problem much simpler.

Because the Garp array remains separate from the main processor, the processor can run at a higher clock rate. This permits Garp to give the best performance on sequential code that has little instruction-level parallelism (ILP).

The VLIW architecture's main relative advantage is that it can exploit ILP outside of loops.

Vector

When the Garp compiler synthesizes a vectorizable loop, the resulting configured array resembles a memory-to-memory vector processor with chained functional units. In Garp, the memory queues stream data into the pipelined data path, and results return directly to memory. The main differences are as follows:

- While vector units can typically chain only two or three operations, the Garp array can "chain" the entire loop body.
- In Garp, feedback loops can be constructed arbitrarily, while vector units can handle only very specialized recurrences—such as sum reduction—if they handle any at all.
- Garp can easily handle data-dependent loop exits, which are are a problem on many vector architectures.

Garp and vector architectures share similarities as well. Both can exploit a great deal of parallelism in loops, while keeping a small and fast scalar processor to handle sequential code.

Superscalar

Superscalar processors can exploit parallelism in code that has been compiled for a sequential processor, and they can adjust their execution dynamically for operations with variable latency. However, the hardware complexity of dynamically determining dependencies between instructions prevents superscalar processors from scaling well beyond a modest number of instruction issue slots. Thus these processors cannot compete with the Garp array in cases with a large amount of exploitable ILP. A Garp-like architecture could use a two- or four-way superscalar processor as its main processor to exploit modest ILP in code not accelerable by its coprocessor.

researchers tackled similar challenges when building compilers for very-long-instruction-word (VLIW) machines. With slight adaptations, we can use very much the same solutions,[3] even though the Garp array's means of execution differs significantly from a VLIW processor's. The main construct we borrow from VLIW compilation is the hyperblock.[4]

For a more detailed description of how Garp compares with other high-performance architectures, see the "Comparing Garp to Other Architectures" sidebar.

The hyperblock

As we use it, a *hyperblock* is formed by joining all the basic blocks along the frequently executed control paths of the loop body, excluding all uncommon paths and problematic code, as shown in Figure 2b. The key idea is that only the hyperblock—containing the common paths—is implemented in the array. When execution takes an excluded path, an *exceptional exit* from the array occurs, and execution continues in software on the main processor. Computation can resume on the array at the start of the next loop iteration. For this approach to be effective, exceptional exits must occur only a small fraction of the time. We use profiling and execution time estimates to intelligently exclude paths from the hyperblock, and also to completely reject loops that do not execute long enough on average to make up for the overhead costs of using the array.

The hyperblock's other key feature is that it increases ILP by merging all the included basic blocks, allowing operations from different basic blocks to be brought together into a single large *dataflow graph* (DFG) and scheduled in parallel, as shown in Figure 2c. In the DFG, nodes represent simple operations, and data edges between nodes indicate data producer-consumer relationships. Basic blocks are merged using *predication*, which eliminates the need for conditional branches. The array performs computation along all included paths, and predicates—Boolean values calculated from the conditions that originally controlled the conditional branches—control multiplexors to select the appropriate values at control merge points. Operations that have side effects external to the array, such as memory stores, have a direct predicate input that enables them only when the particular control path is valid.

The compiler adds *precedence edges* to preserve the original program ordering between each pair of memory operations that might access the same location, unless both are loads. We use array subscript analysis and interprocedural pointer analysis to avoid adding unnecessary precedence edges, which are undesirable because they reduce the amount of parallelism in the dataflow graph.

Unlike the original VLIW hyperblock, our hyperblock contains the loop back edge(s), reflecting that the entire loop executes within the array with no intervention from the main processor. Our hyperblock dataflow graph correspondingly contains cycles

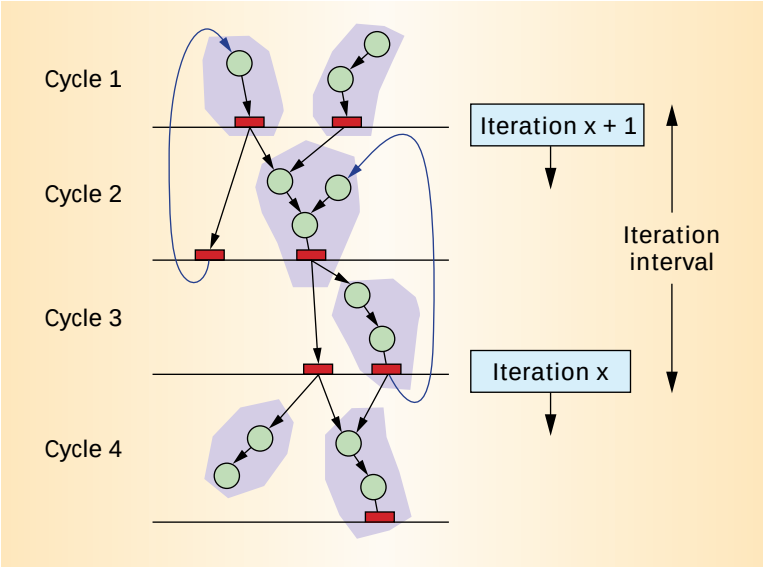


Figure 3. Module-mapped and pipelined DFG. The purple regions indicate the grouping of DFG nodes into modules. Loop-carried edges (blue) limit how closely iterations can follow each other; here, the iteration interval is two cycles. A register (red rectangle) is inserted at every cycle; extra registers are inserted as necessary so that parallel paths have matching delay.

caused by *loop-carried* edges. There can be loop-carried data edges—indicating that a value written in one iteration is read during the next iteration—and loop-carried precedence edges—indicating that an operation in one iteration must execute before another operation in the subsequent iteration.

CONFIGURATION SYNTHESIS

We use a simple and straightforward synthesis approach, performing a direct mapping of nodes in the DFG to modules on the array.

Module mapping and placement

Recall that a single Garp array module can often implement a group of operations from the DFG, usually with just one cycle of latency. It is important that the compiler exploit this capability to make the configuration both smaller and faster.

Module mapping is the task of mapping groups of nodes in the dataflow graph to compound modules in the configuration in a way that minimizes the configuration's size, its critical path, or both. An analogous problem is instruction selection for CISC (complex instruction-set computer) processors: mapping the compiler intermediate representation to complex instructions. Fortunately, an efficient dynamic programming algorithm for CISC instruction selection already exists.[5] We developed a variant of that algorithm to produce efficient data-path modules quickly without resorting to gate-level logic optimization.[6] The speed of this approach is important since a single program compilation may require synthesizing dozens of different kernels.

Next, the compiler decides on a linear placement of the modules in the Garp array, attempting to position connected modules close to one another to reduce the average length of buses and thus make better use of the available intermodule routing resources.

Generating the configuration

After mapping and placement, the compiler must actually construct each module in detail: specify the exact function of each CLB and perform routing internal to the module. The array can implement such a rich variety of functions that it is infeasible to simply instantiate each module by copying it from a static library, as the necessary library would contain tens of thousands of possible modules. Thus our synthesis tool generates all modules on demand. The generator, given a pattern of DFG nodes, the values of constant inputs, input sources, and other data, creates the module.

A simple sequencer is synthesized within the configuration. Its duty is to keep track of the current cycle within the iteration and activate via their control blocks the modules scheduled for that cycle. In particular, a memory operation must execute during the correct cycle. We implement the sequencer in a distributed fashion, with part of it generated inside each module that uses it.

Last, the compiler finalizes the routing between different modules, then generates the configuration bit stream. This bit stream ultimately links in as constant data within the final, complete program's executable file.

ADVANCED TECHNIQUES

The following techniques can enhance Garp's performance.

Speculative loads

One technique to increase effective ILP is to execute loads speculatively, before it is known whether they would have been executed in the original sequential program. Because of this speculation, such loads sometimes execute with invalid virtual addresses. The Garp hardware supports speculative execution of loads in that case by simply ignoring invalid virtual address exceptions and returning arbitrary data. Ignoring these exceptions can make debugging more difficult but cannot cause a correct program to fail. In contrast, speculative loads from virtual addresses that *are* valid but not currently resident still cause a trap to the operating system so that it can service the page fault.

Pipelining

The pipelining process overlaps the execution of different iterations, as shown in Figure 3, reducing the total time for all iterations to complete. The smaller the *iteration interval*—the delay between successive iterations—the more overlap and the greater the performance gain. Loop-carried dependencies can limit the amount of overlap, since an iteration must wait for the loop-carried value produced by the preceding iteration. Also, care must be taken that memory accesses from overlapping iterations do not attempt to use the address bus simultaneously; these conflicts may limit the amount of overlap as well.

Table 1. Kernels from a wavelet image compression program.

Kernel	Percentage of original software execution time	Iteration Interval	No. of queues used	ILP (average operations per nonstall cycle)	No. of executions	Average no. of compute cycles per execution (including stalls)	Average no. of overhead cycles per execution	Net speedup over MIPS only
forward_wavelet_696	18.2	2	2	10.0	448	1,176	114	2.1
forward_wavelet_647	13.8	2	2	10.0	448	310	91	5.1
init_image_354	12.8	1	2	8.0	1	65,852	564	12.7
forward_wavelet_711	10.1	2	2	7.0	448	241	59	4.9
entropy_encode_544	10.0	1	1	5.0	1	65,538	989	9.9
forward_wavelet_674	9.3	1	3	13.0	448	128	76	6.6
block_quantize_411	5.5	2	0	5.5	320	353	56	2.8
entropy_encode_557	3.9	6	0	2.8	3,262	31	24	1.4
RLE_encode_509	3.8	1	1	11.0	774	22	48	4.6

Garp's support for speculative loads is crucial for pipelining because an iteration will begin execution before it is known whether the previous iteration causes a loop exit. If Garp did not have speculative loads, a kernel would have to evaluate the exit condition at the end of one iteration before a load at the start of the next iteration could execute, greatly inhibiting iteration overlap.

Memory queues

Many applications for reconfigurable computing operate on contiguous data streams. The memory accesses for such streams can often be fully overlapped with computation by buffering and reading ahead, writing behind, or both. We could have implemented this buffering activity in Garp's array, but we felt it better to provide dedicated hardware for this common task; doing so frees up more array resources for the actual computation. Designers have considered similar mechanisms for standard processors.[7]

Garp has three memory queues supporting sequential streams. The main processor initializes the queues with a starting address and data size before array execution begins. From the array's perspective, queue accesses resemble other memory accesses except that the array does not provide the address. Read response takes less time because the data is already waiting in the queue buffer. Unlike random memory accesses, the accesses to all three queues can occur every clock cycle over three independent memory data buses. Each queue can optionally be configured as non-cache-allocating, so that streaming data used only once does not pollute the cache.

The compiler tries to use the memory queues as much as possible. The first necessary condition is unit stride: The address increment must match the data size, whether it is one, two, or four bytes. The increment and access need not occur every iteration, but the two operations must always occur together. Finally, it must be legal to prefetch the load or delay the store.

SIMULATION RESULTS

We gathered results using a cycle-accurate simulator of the entire Garp chip with a 32-row array and a memory system modeled after that of the Ultrasparc processor. The simulator models all stalls from cache misses and interlocks, but does not attempt to model multiple processes or to accurately time operating-system activity.

Wavelet image compression

The first benchmark we consider, wavelet image compression,[8] stresses reconfigurability by splitting execution time among several kernels. For each kernel, Table 1 provides a comparison of its execution on the MIPS processor versus on the array for processing a 256×256 pixel image. The table gives results for net speedup, factoring in configuration and data transfer overhead. These kernels capture 87 percent of the original software execution time. Overall speedup is 2.9 versus MIPS only, compiled with `gcc -O3`.

Kernels that execute only once show a large overhead per use because their single execution bears the entire cost of loading the configuration. On the other hand, kernels that execute hundreds or thousands of times can amortize the configuration load cost across all executions. Thanks to Garp's configuration cache, in most cases a configuration loads from the off-chip secondary cache only once even though the application switches among different kernels. For kernels that executed many times, data transfer rather than configuration contributed most to the overhead cost.

For many of these loops, the use of memory queues boosts throughput significantly. The forward_wavelet kernels each perform four memory accesses per iteration. Without the memory queues, contention for the

Table 2. Garp's speedups over Ultrasparc for hand-coded functions.

Function	Data size	Speedup	Limiting factor
Image median filter	640 × 480 pixels	43	Compute throughput
DES (ECB mode)	1 Mbyte	18.7	Compute throughput
Image dithering	640 × 480 pixels	17.0	Compute throughput
strlen	1,024 chars	14.2	Memory bandwidth
strlen	16 chars	1.84	Overhead
Sort	2 Mbytes	2.2	Scattered memory accesses

address bus would limit the maximum throughput to one iteration every four cycles. With the queues, multiple accesses can occur each cycle, reducing the iteration interval to two or even one cycle.

The prefetching action of load queues is also significant. The top two kernels are very similar except that forward_wavelet_696 performs two random loads while forward_wavelet_647 performs two queue loads. Even though they both have nominal iteration intervals of two cycles, forward_wavelet_696 experiences many data cache stalls, reducing its overall performance.

Gzip compression

We also studied compilation of the gzip compression program, which proved challenging due to its irregular memory accesses. In many cases, these irregularities reduced the amount of parallelism and prevented the use of pipelining or memory queues. Also, typically each loop executed for fewer cycles each time, making overhead costs more significant. Gzip spent about half its execution time in the array. The best kernel saw a net speedup of a factor of 2, but ran only a small fraction of the execution time. Although the array accelerated each kernel individually by some amount, the use of the array interfered with global optimization of the remaining software portion of the program, negating the benefit of using the array.

Compilation time and code expansion

Compilation time for Garp, including synthesis of all array configurations, is typically much less than double that of compiling to just the software processor using SUIF. This compilation speed, much faster than traditional hardware synthesis techniques, results from our adoption of software compilation algorithms. The size of the compiled wavelet benchmark grew by 16 percent after the addition of code and configuration data for using the reconfigurable array. Typical expansion ranges from 10 to 50 percent although relatively small changes in the configuration format could greatly reduce this factor. Specifically, given the repetitive bit slice data paths, run length encoding could often be effective.

Garp versus Ultrasparc

From an architectural point of view, we must ask if the reconfigurable array is a wise use of transistors, given other architectural alternatives. To help answer this question, we compare Garp to a four-way superscalar Ultrasparc 170 processor. To enable direct comparisons, we modeled the simulated Garp memory system after the Ultrasparc's. If implemented using the same VLSI process as the Ultrasparc, the Garp chip would be roughly the same size: Garp has room for the array because it has a smaller single-issue integer unit and no floating-point unit. The reconfigurable hardware and its memory interface fills 10.5 mm × 9.5 mm in the Ultrasparc's 0.5-µm process, approximately a third of the die. The Ultrasparc runs at 167 MHz, while we estimate that Garp's implementation would run at 133 MHz. We calculated relative performance using execution time, not cycles.

We compiled and ran the benchmarks on Ultrasparc to compare against our automatic compilation path to Garp. We used identical source code for both targets. The Ultrasparc executable did not utilize Visual Instruction Set (VIS) multimedia instructions because the Ultrasparc compiler did not automatically generate them. In fact, automatic VIS compilation techniques would benefit the Garp compilation path equally since the array could implement segmented modules. We excluded file I/O because of the difficulty in accounting for its variability.

For the wavelet image compression benchmark, Garp, using the array, ran 68 percent faster than the Ultrasparc, which in turn ran 73 percent faster than the Garp MIPS processor alone. Garp's significant performance increase, even with its lower clock rate, is due to the amount of ILP it can exploit, sustaining close to 10 or more operations per nonstalled cycle in many cases, compared to Ultrasparc's best-case maximum of four operations per cycle.

For the gzip benchmark, the superscalar Ultrasparc ran just 14 percent faster than Garp using the array, which performed the same as just the single-issue Garp MIPS. For most loops the limiting factor was memory latency, so that neither the Ultrasparc's nor the Garp array's ability to exploit ILP was very useful.

Hand-coded examples

In a production system, programmers would likely use automatic compilation in conjunction with hand-coded libraries of common and domain-specific functions. Table 2 shows Garp's speedups over the Ultrasparc for some hand-mapped examples to give an idea of Garp's potential in this situation. In all cases, we assume that the configuration cache already holds the configuration. Although the Ultrasparc versions we compare against do not use VIS extensions, we estimate that doing so would make a significant

difference only in dithering. In that one case, the difference would be less than a factor of two.

The Garp project shares similar goals with previous projects, such as the Programmable Reduced Instruction Set Computer (PRISC)[9] and National Adapative Processing Architecture (NAPA),[10] but each project has chosen a unique design and varying degrees of automatic compilation. Our compiler's predication approach resembles that used in the PRISM compiler,[11] an earlier project that investigated semiautomatic C compilation to an off-chip reconfigurable coprocessor. Finally, the Synopsys Nimble Compiler project,[12] building on the Garp compiler presented here, is researching retargetability and further optimizations for embedded applications.

Our results show that Garp's features can be effectively utilized through automatic compilation. Perhaps most importantly, in many cases the compiler used Garp's memory queues to provide high-bandwidth, low-latency data access for the array. Without this capability, the compiler's ability to produce high-throughput, optimized configurations would often be wasted.

The most important remaining work is the study of Garp and its compiler across a broader range of benchmark applications. Our future findings will direct the development of new optimizations to the Garp compiler and help us draw more conclusions about the strengths and weaknesses of the Garp architecture. ❖

References

1. J. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, K.L. Pocek and J.M. Arnold, eds., IEEE CS Press, Los Alamitos, Calif., 1997, pp. 12-21.
2. R. Wilson et al., "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers," *SIGPLAN Notices*, Dec. 1994, p. 31; also available online at http://suif.stanford.edu.
3. T. Callahan and J. Wawrzynek, "Instruction-Level Parallelism for Reconfigurable Computing," *Proc. 8th Int'l Workshop Field-Programmable Logic and Applications*, Springer-Verlag, Berlin, 1998, pp. 248-257.
4. S. Mahlke et al., "Effective Compiler Support for Predicated Execution Using the Hyperblock," *Proc. 25th Int'l Symp. Microarchitecture*, IEEE CS Press, Los Alamitos, Calif., 1992, pp. 45-54.
5. C. Fraser, D. Hanson, and T. Proebsting, "Engineering a Simple, Efficient Code-Generator Generator," *ACM Letters on Programming Languages and Systems*, Sept. 1992, pp. 213-226.
6. T.J. Callahan et al., "Fast Module Mapping and Placement for FPGAs," *Proc. ACM/SIGDA Int'l Symp. Field Programmable Gate Arrays*, ACM Press, New York, 1998, pp. 123-132.
7. S. McKee et al., "Smarter Memory: Improving Bandwidth for Streamed References," *Computer*, July 1998, pp. 54-63.
8. S. Kumar, "Stressmark Adaptive Computing Systems Benchmarks," http://www.htc.honeywell.com/projects/acsbench/.
9. R. Razdan and M.D. Smith, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units," *Proc. 27th Ann. Int'l Symp. Microarchitecture*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 172-180.
10. C. Rupp et al., "The NAPA Adaptive Processing Architecture," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 28-37.
11. P.M. Athanas and H.F. Silverman, "Processor Reconfiguration through Instruction-Set Metamorphosis," *Computer*, Mar. 1993, pp. 11-18.
12. Y. Li et al., "Hardware-Software Co-Design of Embedded Reconfigurable Architectures," *Proc. 37th ACM/IEEE Design Automation Conference*, ACM Press, New York, to appear June 2000.

Timothy J. Callahan is a PhD student in electrical engineering and computer science at the University of California, Berkeley. His research interests include compiler and microarchitecture techniques for exploiting instruction-level parallelism, electronic design automation, and hardware-software codesign. He received an MS in computer science from UC Berkeley and is a member of the IEEE Computer Society.

John R. Hauser is a PhD student in computer science at the University of California, Berkeley. His research interests include computer architecture, compilers, computer arithmetic, and exception handling. He received an MS in computer science from UC Berkeley.

John Wawrzynek is a professor of electrical engineering and computer science at the University of California, Berkeley. His research interests include reconfigurable computing, VLSI systems design, computer architecture, and computer music. He received a PhD in computer science from the California Institute of Technology and is a member of the IEEE.

Contact Callahan at timothyc@cs.berkeley.edu.