Instruction-Level Parallelism for Reconfigurable Computing

Timothy J. Callahan and John Wawrzynek

University of California at Berkeley, Berkeley CA 94720, USA
http://www.cs.berkeley.edu/projects/brass/

Abstract. Reconfigurable coprocessors can exploit large degrees of instruction-level parallelism (ILP). In compiling sequential code for reconfigurable coprocessors, we have found it convenient to borrow techniques previously developed for exploiting ILP for very long instruction word (VLIW) processors. With some minor adaptations, these techniques are a natural match for automatic compilation to a reconfigurable coprocessor. This paper will review these techniques in their original context, describe how we have adapted them for reconfigurable computing, and present some preliminary results on compiling application programs written in the C programming language.

1 Introduction

In this work we consider compilation for a hybrid reconfigurable computing platform consisting of a microprocessor coupled with field-programmable gate array (FPGA) circuitry used as a reconfigurable accelerator. The FPGA is configured to provide a customized accelerator for compute-intensive tasks. This acceleration results in part from the parallel execution of operations, since the FPGA has no von Neumann instruction fetch bottleneck. The microprocessor is used for "random" control-intensive application code and for system management tasks. It also provides binary compatibility with existing executables, which eases the migration path to reconfigurable computing [4]. We assume support for rapid run-time reconfiguration to allow several different tasks in the same application to be accelerated [5].

For ease of programming these systems, it is best if a single, software-like language is used for describing the entire application, encompassing computation on both the microprocessor and the FPGA. But traditional imperative software languages are basically sequential in nature; starting from there, it is a challenging task to exploit the reconfigurable hardware's parallel nature. Previous efforts have corrected this mismatch by using languages with constructs to explicitly specify either data parallelism [8, 9] or more general parallelism [15, 2, 16]. However, the farther such a language's semantics deviate from those of sequential languages, the more difficult it is to train programmers to use it efficiently, and the more work is involved in porting "dusty deck" sequential code to it.

This work instead investigates automatic extraction and hardware compilation of code regions from dusty deck C code. While it is unlikely the resulting performance will be as good as if a human rewrote the code in a parallel style, this approach has its advantages: (i) it gives immediate performance benefit from reconfigurable hardware with just a recompilation, and (ii) it is useful in cases in which the execution time is spread across more kernels than are worth recoding by hand.

Automatic compilation of sequential code to hardware poses several challenges. In such code, basic blocks are typically small so that little instruction-level parallelism is found within each one. Also, operations difficult to implement directly in hardware, such as subroutine calls, are often sprinkled throughout the code. Finally, loops often contain conditionals with rarely executed branches that interfere with optimization.

These features of sequential code similarly caused problems for VLIW machines. A key technique used by VLIW compilers to overcome these obstacles was to optimize the common execution paths to the exclusion of all rarely executed paths. Applying these same techniques allows us to accelerate loops that could not otherwise be mapped to the coprocessor due to an operation that is infeasible to implement in hardware. Without the exclusion ability, an infeasible operation on even an uncommon path would prevent any of the loop from being accelerated. Furthermore, by excluding uncommon paths, the remaining paths typically execute more quickly, and in the case of reconfigurable computing, less reconfigurable hardware resources are required.

Our approach is greatly influenced by our target platform, the theoretical Garp chip [11]. The Garp chip tightly couples a MIPS microprocessor and a datapath-optimized reconfigurable coprocessor. The coprocessor is rapidly reconfigurable, making it possible to speed up dozens or hundreds of loops. If a desired configuration has been used recently it is likely still in the *configuration cache* and can be loaded in just a few cycles. The flipflops in each row of the Garp array are accessible as a single 32-bit coprocessor register; transfers between such a coprocessor register and a microprocessor register can be performed at a rate of one per clock cycle. The coprocessor can directly access the main memory system (including data cache) as quickly as can the microprocessor. Because of these features, the Garp coprocessor can accelerate many loops that would be impractical to implement with a more traditional FPGA coprocessor. It is also due to these features that we can easily off-load uncommon computations to the main processor, allowing the common cases to execute faster on the coprocessor.

2 VLIW Background

VLIW processors possess a number of functional units that can be utilized in parallel, connected to each other through a multiported register file. Builders of such machines long ago encountered difficulty extracting enough ILP from sequential programs to keep all of the functional units busy. Studies have shown that amount of ILP within a basic block is typically only 2–3.5 [7]. *Trace scheduling* [6] is a technique to enhance the amount of ILP by combining sequences of basic blocks along the most commonly executed linear path. This forms a larger unit for scheduling called a trace, within which more ILP is available. While this approach works well when there is a single dominant sequence, it is less successful when there are multiple paths that are executed with comparable frequency. The *hyperblock* [14] was developed to address this situation, and is the central structure we borrow from VLIW compilation.

A hyperblock is formed from a contiguous group of basic blocks, usually including basic blocks from different alternative control paths. For optimization purposes, a hyperblock will typically include only the common control paths and not rarely taken paths. By definition, a hyperblock has a single point of entry – typically the entry of an inner loop – but may have multiple exits. An exit may jump back to the beginning of the same hyperblock (forming a back edge of the loop); an exit may jump to an excluded path of the loop; or an exit may continue to other code, as in the case of a normal loop exit. Some basic blocks may need to be duplicated outside of the hyperblock in order for the single-entry constraint to be obeyed. Fig. 1 shows an example hyperblock.

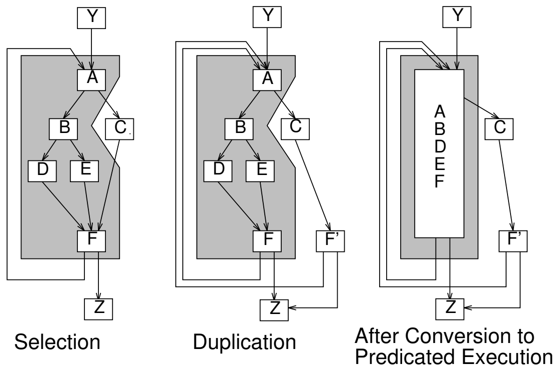

Fig. 1. Hyperblock formation for VLIW compilation.

The basic blocks selected to form a hyperblock are merged by converting any control flow between them to a form of *predicated execution*. We consider specifically *partially predicated execution* [13]. With partially predicated execution, the instructions along *all* of the included control paths are executed unconditionally (with the exception of memory stores), and *select* instructions are inserted at control flow merge points to select the correct results for use in subsequent computation (Fig. 2). If a select instruction's first operand (called a *predicate*) is true, the result is the value of the second operand, otherwise the result is the value of the third operand. The predicate is an expression of the boolean values that originally controlled conditional branches. The result is essentially speculative execution of all included paths; although this approach rapidly consumes resources, it also gives the best performance since it exposes the most ILP and reduces critical path lengths.

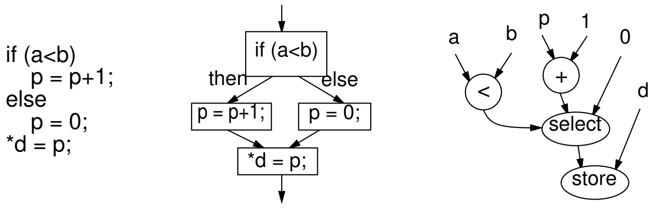

Fig. 2. C code, control flow graph, and partially predicated execution.

The challenge for the VLIW compiler is deciding which basic blocks should be included in each hyperblock. In general, rarely taken paths should be excluded. Excluding them allows the common cases to achieve higher performance, at the expense of causing the rare paths to execute more slowly due to the branch penalty of exiting a hyperblock. Factors considered when selecting the basic blocks to compose a hyperblock include relative execution frequencies, resource limitations, and the impact on the critical path.

3 Using the Hyperblock for Reconfigurable Computing

Our compiler uses the hyperblock structure for deciding which parts of a program are executed on the reconfigurable coprocessor as well as for converting the operations in the selected basic blocks to a parallel form suited to hardware. Being able to exclude certain paths from implementation on the reconfigurable coprocessor is very useful. Besides making the configuration smaller and often faster, it allows us to ignore certain operations that are impossible to implement using the coprocessor, e.g., subroutine calls, as long as they are on uncommon paths. If a loop body had to be implemented on the coprocessor as an all-or-nothing unit, the existence of a subroutine call even on an uncommon path would exclude the entire loop from consideration for acceleration using the coprocessor.

For each accelerated loop we form one hyperblock, which will be the portion of the loop body that is implemented on the reconfigurable coprocessor. Hyperblock exits are points where execution is transferred from the reconfigurable coprocessor back to the main processor. Each hyperblock eventually becomes one coprocessor configuration, and thus there is one configuration per accelerated loop. The partially predicated execution representation maps directly to hardware, with the select instructions implemented as multiplexors. The resulting dataflow graph is very similar to the "operator network" used in the PRISM-II compiler [1].

One adjustment in our use of the hyperblock is that a loop back edge from an exit directly back to the top of the hyperblock is considered to be *internal* to the hyperblock. This reflects the fact that normal loop iteration control is performed on the reconfigurable coprocessor with no intervention from the main processor. Thus such a back edge is not actually considered an exit. The remaining exits can be classified as either finished or exceptional. *Finished* exits are those that are taken because the loop has finished all its iterations (e.g., Exit 1 in Fig. 3). *Exceptional* exits are those that are taken because an excluded basic block must be executed (e.g., Exit 2 in Fig. 3). After an exceptional exit is taken, the hyperblock may be reentered at the beginning of the next iteration.

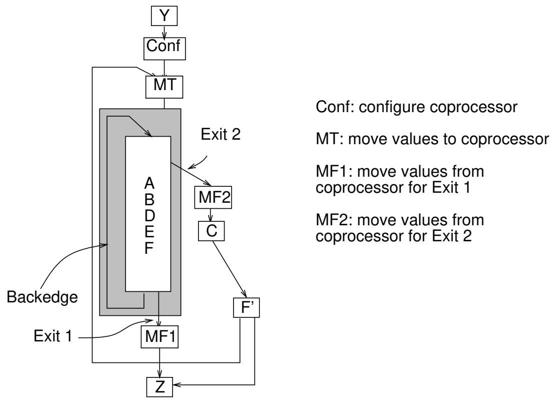

Fig. 3. Hyperblock formation for compilation to reconfigurable coprocessor. The hyperblock ABDEF is executed using the reconfigurable coprocessor, while all other basic blocks are executed on the microprocessor.

4 Execution Model

The unit of code that gets accelerated on the coprocessor is a loop[1]. General loop nesting is not currently supported. Thus accelerated loops cannot contain any inner loops on included paths. A loop that contains inner loops only on *excluded* paths can still be accelerated.

When an accelerated loop is reached in the program, the processor activates the correct configuration for the coprocessor, moves initial data into FPGA registers, and starts the coprocessor. The processor then suspends itself and does not wake up until the coprocessor is finished.

As the coprocessor executes, a simple sequencer keeps track of the current cycle in the iteration. The sequencer activates functional units that are scheduled for a specific cycle. In particular, a memory operation must execute during the correct cycle; during other cycles its address input may be invalid.

The sequencer is also used to determine when the end of the iteration has been reached. The number of cycles in the iteration is determined by the critical path through the hyperblock and is the same every iteration. At the end of an iteration, loop-carried values are latched for use in the next iteration.

[1] The execution model presented here is specific to our automatic compilation. It should not be inferred that this is the only model allowed by the Garp architecture.

Coprocessor execution will continue indefinitely until an exit is taken. An exit can occur at any cycle of an iteration, and a hyperblock may have several different exits. When any exit is activated, a builtin feature of the Garp reconfigurable array is used to halt the array instantaneously, freezing all values in registers. This action also awakens the processor.

Once the processor awakes, it must determine which exit was taken from the hyperblock. Once done, it moves any live values from the coprocessor back to the main registers. This transfer depends on which exit was taken, as there are likely different sets of live variables at different exits, and also different values for a given variable at different exits. If an exceptional exit was taken, the remainder of the iteration is completed on the processor, and at the start of the next iteration control is again transferred to the coprocessor.

5 Compiler Flow

Front End. We have used the SUIF compiler system [10] to implement our prototype C compiler targeting the Garp chip. SUIF's standard C parser and front end optimizations are utilized. A basic block representation of the code is then constructed using a control flow graph library [12]. This library includes natural loop analysis routines that allow us to recognize and process any kind of loop, even one formed by a backwards goto statement in the original source code.

Hyperblock Selection. At each loop an attempt is made to form a hyperblock. Initially all of the basic blocks in the loop are marked as included, and then blocks are systematically excluded. A basic block can be excluded from the hyperblock (i) because it contains an infeasible operation (floating-point, division, remainder, or subroutine call), (ii) because it is the entry of an inner loop (which will have its own hyperblock), (iii) because it needs to be excluded in order for the configuration to fit on the available resources, or (iv) simply to improve the performance of the remaining hyperblock. Currently (i) and (ii) are operational, and heuristics for (iii) and (iv) are under development.

The exclusion of one block often implies the exclusion of other blocks due to the *trimming* process. "Dead end" basic blocks – those blocks that ultimately lead only to excluded blocks – are trimmed. We also trim "unreachable" blocks – those that would only be reached by going through an excluded block. After this trimming we know that every remaining basic block is on a cycle contained completely in the hyperblock.

Hyperblock Duplication. We find it useful to duplicate all basic blocks selected for the hyperblock, retaining the complete original non-hyperblock version of the loop as well. This is in contrast to the VLIW approach, which only duplicates basic blocks as necessary to avoid re-entering the hyperblock. For our purposes, the non-hyperblock version is the "software" version of the loop, while the hyperblock copy is the "hardware" version. At an exceptional exit, control flows out of the hardware version and continues at the corresponding point in the software version.

The hardware version of the loop is a temporary construction; its presence aids in the construction of the interface instructions and the dataflow graph. Ultimately, however, the basic blocks in the hardware version will be deleted, since that computation will actually be performed in the FPGA.

Interface Synthesis. At the points where the hardware version of the loop interfaces with the rest of the program, interface code must be synthesized. This includes the instructions to load the correct configuration, to move values back and forth between the coprocessor and main processor registers, and to determine which exit was taken when the coprocessor finishes. When the data transfer code is synthesized, live variable analysis is used to avoid unnecessary transfers.

Dataflow Graph Formation. The computation in the basic blocks of the duplicated hyperblock is converted to partially predicated execution form as was shown in Fig. 2. This creates a dataflow graph where all of the control flow has been converted to explicit data transfer. This form maps very directly to the reconfigurable hardware. The "select" instructions are implemented in hardware as multiplexors and are analogous to the "merge-muxes" used in PRISM-II.

A number of optimizations are performed on the dataflow graph before it is written out for synthesis. The number of multiplexors can often be reduced at the expense of a small amount of extra boolean logic. Also, boolean signals (e.g., the results of comparisons) are identified and used to simplify multiplexors to gates, and to simplify comparisons to no-ops or inversions. We found that these cases arise frequently due to complex logical expressions typically found in C code.

The dataflow graph may contain multiple memory access operations. If it cannot be determined at compile time whether or not two memory operations access the same location, then they must be executed in the same order as they appear in the sequential program input, unless they are both loads. Such orderings are preserved using precedence edges in the dataflow graph; these edges indicate that the source node must execute before the destination node. Precedence edges are also needed between exits and memory writes to assure that writes don't occur when they shouldn't, or not occur when they should.

Synthesis. Each dataflow graph is written out and fed to the GAMA datapath synthesizer [3] to create the configuration for the coprocessor. GAMA looks for opportunities to merge neighboring operations (for example, an addition and an exclusive-or) to form a compound functional unit when this is smaller and/or faster than implementing each individually. GAMA also synthesizes a sequencer that counts the cycles in each iteration and activates modules appropriately.

Final Compilation and Linking. Assuming the synthesis is successful, the code is patched to direct control to the hardware version of the loop. The modified code is converted from SUIF back to C and cross-compiled with a modified version of `gcc` that understands Garp's extensions to the MIPS instruction set. The final executable links in the software object files along with the coprocessor configurations, which are in the form of integer array initialization data.

Execution. As the hardware is not currently available, the executable is run on the Garp simulator, which accurately models cache misses, configuration loading delays, and other important features to provide an accurate prediction of execution behavior.

6 Preliminary Results

The two primary benefits of the approach presented here are (i) increasing the performance of computation on the coprocessor, and (ii) increasing the fraction of a program that can be accelerated using the coprocessor. We will not be able to evaluate the magnitude of the first benefit until we complete development of heuristics for excluding basic blocks to get better performance. However, we have collected preliminary data regarding the second benefit – excluding parts of a loop that can't be implemented in hardware in order to allow the remainder of the loop to be accelerated.

The results are presented in Table 1. Execution cycles are classified into one of 5 categories, and the cumulative time in cycles in each category is reported for four application/dataset combinations. The categories are as follows:

- **Single Exit Loops** have a single exit and contain no infeasible operation.
- **Multi Exit Loops** have multiple exits and contain no infeasible operations.
- **Hyperblock Loops** have excluded blocks due to infeasible operations.
- **Unfruitful Loops** execute for too few cycles per exit to overcome overhead for using the coprocessor. We estimate a factor of two speedup using the coprocessor, and 25 cycles of overhead per exit. Thus loops that executed 50 cycles or fewer per exit on average would be slowed down using the coprocessor and fall into this category. This is admittedly a rough guess.
- **Other** Cycles not included above – straight-line code, code in library routines, and code in infeasible loops (loops that have an infeasible operation or inner loop on every path through their body).

With the `gzip` examples, we see that a large fraction of computation cycles are captured in the first two relatively simple classes of loops. This indicates that most loops in `gzip` don't contain infeasible operations. The hyperblock's ability to exclude some paths helps only a small degree.

With the `cpp` examples, however, the hyperblock approach allows a significant increase in the amount of computation that could be accelerated using the coprocessor. This is because many loops in `cpp` contain subroutine calls, e.g., to report errors. The three top time-consuming loops in `cpp` all contained infeasible operations, but in two of them, the infeasible operations were *never* executed, and in the third they were only rarely executed. This gives anecdotal support to the intuitive feeling that infeasible operations often occur on rarely executed paths.

Table 1. Execution time breakdown in cycles. Categories explained in text.

Test case	Single Exit Loops	Multi-exit Loops	Hyperblock Loops	Unfruitful Loops	Other	Total
`gzip` C source	530149 33.1%	586173 36.6%	143449 9.0%	134213 8.4%	209459 13.1%	1603443 100.0%
`gzip` English text	601187 28.6%	662164 31.5%	218534 10.4%	179781 8.6%	439624 20.9%	2101290 100.0%
`cpp` input 1	2949104 20.2%	2158983 14.8%	8423327 57.6%	213459 1.5%	878407 6.0%	14623280 100.0%
`cpp` input 2	1092072 18.5%	894918 15.2%	2179589 37.0%	265824 4.5%	1463763 24.8%	5896166 100.0%

7 Summary

We have adapted the hyperblock from VLIW compilation for our use in compiling to a reconfigurable coprocessor. Commonly-executed basic blocks are combined to form a large hyperblock, exposing instruction-level parallelism and allowing speculative execution to achieve high performance using the reconfigurable hardware. By excluding rarely-executed basic blocks, the compiler can produce configurations are smaller and faster, and can accelerate a greater number of loops than would be possible otherwise. Preliminary results show that in some cases our approach significantly increases the fraction of execution cycles that can be accelerated using the coprocessor when compiling dusty-deck C code.

8 Future Work

In future work we will develop hyperblock formation heuristics that integrate path profiling information and hardware estimation. We can then evaluate the performance benefit from intelligently excluding computation from the accelerated loop on the coprocessor.

9 Acknowledgements

This work benefited from discussions with others in the BRASS Research Group including John Hauser and André Dehon. We thank Randy Harr of Synopsys and the anonymous reviewers for their comments on earlier drafts of this paper.

This work is supported in part by DARPA grant DABT63-96-C-0048, DARPA/AFRL grant F33615-98-2-1317, Office of Naval Research grant N00014-92-J-1617, and National Science Foundation grant CDA 94-01156.

References

[1] L. Agarwal, M. Wazlowski, and S. Ghosh. An Asynchronous Approach to Efficient Execution of Programs on Adaptive Architectures Utilizing FPGAs. In *Proceed-*

ings IEEE Workshop on FPGAs for Custom Computing Machines, pages 101–10. IEEE Comput. Soc. Press, 1994. AN4754552.

[2] M. Aubury, I. Page, G. Randall, J. Saul, and R. Watts. Handel-C Language Reference Guide.

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

[4] A. DeHon. DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century. In *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, pages 31–9. IEEE Comput. Soc. Press, 1994.

[5] J. G. Eldredge and B. L. Hutchings. Density Enhancement of a Neural Network Using FPGAs and Run-Time Reconfiguration. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 180–188, Napa, CA, Apr. 1994.

[6] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, Cambridge, MA, 1985.

[7] J. A. Fisher and B. R. Rau. Instruction-Level Parallel Processing. In H. C. Torng and S. Vassiliadis, editors, *Instruction-Level Parallel Processors*, pages 41–49. IEEE Computer Society Press, 1995.

[8] M. Gokhale and B. Schott. Data parallel C on a reconfigurable logic array. *Journal of Supercomputing*, pages 1–24, 1994.

[9] S. Guccione and M. Gonzalez. A Data-Parallel Programming Model for Reconfigurable Architectures. In *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, pages 79–87. IEEE Comput. Soc. Press, 1993.

[10] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, Dec. 1996. See also http://suif.stanford.edu/.

[11] J. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, Napa, CA, Apr. 1997.

[12] G. Holloway and C. Young. The Flow Analysis and Transformation Libraries of Machine SUIF. In *Proceedings of the Second SUIF Compiler Workshop*, Aug. 1997. Available from http://www-suif.stanford.edu/suifconf/suifconf2/.

[13] S. Mahlke, R. Hank, J. McCormick, D. August, and W. Hwu. A Comparison of Full and Partial Predicated Execution Support for ILP Processors. In *Proceedings 22nd Annual International Symposium on Computer Architecture*, pages 138–49, Santa Margherita Ligure, Italy, June 1995. ACM.

[14] 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 International Symposium on Microarchitecture*, pages 45–54, Dec. 1992.

[15] I. Page and W. Luk. Compiling occam into FPGAs. In *FPGAs. International Workshop on Field Programmable Logic and Applications*, pages 271–283, Oxford, UK, Sept. 1991.

[16] N. Wirth. Hardware Compilation: Translating Programs into Circuits. *IEEE Computer*, 31(6):25–31, June 1998.