

Kernel Formation in Garpcc

Tim Callahan

1. Introduction

The Garp project [3] quantitatively investigates the benefits of adding an on-chip dynamically reconfigurable coprocessor to a standard instruction processor. Intended for acceleration of loops, Garp's coprocessor performs iteration control and both streaming and random memory accesses without assistance from the instruction processor.

The companion project Garpcc [2] investigates whether new compilation approaches can enable automatic exploitation of the coprocessor starting from standard C code. No hints regarding hardware/software partitioning are expected, although profiling data is assumed.

A key technique used by Garpcc is to exclude rarely-taken control paths from the coprocessor implementation of the loop [1]; when an iteration takes an excluded path, control hops back to the instruction processor to execute the remainder of that iteration, and control returns to the coprocessor at the start of the next iteration. The set of included paths is called the *kernel* of the loop.

This approach of excluding some paths but including others closely parallels hyperblock formation in VLIW compilation as described by Mahlke et.al. [5], especially since predicated execution is used to expose parallelism across the included paths. Garpcc builds a dataflow graph (DFG) using an aggressively speculative form of predication where safe operations along all included paths are performed unconditionally, and at control flow merge points, multiplexors controlled by predicates select the appropriate values for further use. Only non-safe operations (stores and exits) are directly guarded by predicates. The synthesis step from DFG to hardware utilizes a fully spatial approach (no sharing of function units / modules).

Despite similarities with VLIW hyperblock formation, Garpcc's framework and decision process for choosing included paths differs significantly from that described by Mahlke.

2. Kernel Formation Framework

When forming each kernel, Garpcc uses a subtractive, iterative approach: it starts with the entire original loop and

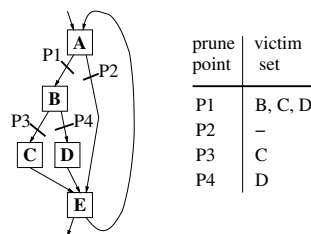


Figure 1. Prune points.

then repeatedly removes portions out of necessity and/or to improve performance of the kernel. This differs from Mahlke's additive hyperblock formation, which starts with just the most common path and then adds additional parallel paths while deemed beneficial.

Basic blocks cannot be removed in arbitrary groups, or else the remaining kernel may contain unreachable and/or "dead-end" blocks. Obviously unreachable blocks make no sense. Dead end blocks, those executed only en route to an exit, are considered, as are exits, to be an exceptional case, and thus should not consume reconfigurable resources.

Garpcc uses the concept of *prune points* to identify related groups of paths / basic blocks to be considered together for elimination while still leaving a sensible kernel. A legal prune point is either outgoing edge of a basic block that has two successor blocks both in the (current) kernel. See Figure 1. Each prune point has an associated *victim set*: those basic blocks that become unreachable once the prune is performed. The victim set may be empty as with P2 in Figure 1; this may still be a beneficial prune due to the eliminated multiplexor area and delay.

After a prune is applied (i.e. removing its victim set), the remaining kernel is still sensible; in particular, the source of the prune point still has a successor within the kernel, and therefore is not a dead end block. Likewise no unreachable basic blocks remain.

3. Evaluating a Potential Prune

Given an existing DFG representing the current kernel, it is much easier to estimate the effect of *removing* basic blocks than *adding* basic blocks. This is one of many fac-

tors that motivate the subtractive approach. To evaluate a prune, Garpcc temporarily flags the blocks in the prune's victim set as "hidden"; then via cross-pointers, corresponding nodes in the DFG are also (temporarily) flagged as "hidden". Any multiplexor having one data input arriving via a hidden basic block is flagged as "short-circuited", acting as a direct connection from its other data input to its consumers. Estimation routines ignore hidden nodes and treat short-circuited multiplexors as having no area or latency.

Once the DFG nodes are appropriately flagged, estimations for area, critical path, and critical cycle are reevaluated. The new estimations and appropriately redistributed execution counts from profiling are used to give an estimate for total execution time for the potential kernel. Overhead costs are also quickly evaluated: refilling the hardware pipeline, transfer of live variables to/from coprocessor registers, execution time in software after kernel exits, etc. All factors are considered to give an overall time of execution for a loop with the given prune: hardware execution, software execution, and transfer overhead. After evaluation, the flags are removed, and evaluation of the next prune candidate is performed.

Due to the direct, spatial implementation of the DFG into hardware, estimates of area and delay for coprocessor execution are quite precise. There are some sources of imprecision of course: downstream mapping to optimized modules; and placement, which effects inter-module delay in the kernel hardware implementation. Yet, this is a much more direct effort to estimate kernel performance compared to the weighted heuristics used in the VLIW hyperblock formation. Of course, our effort increases compilation effort, but we argue it is worthwhile given the reduction of compilation time with today's processors. Increased compilation time is especially warranted in the domain of embedded processors.

4. Flow

Garpcc performs pruning in three phases. After each phase, checks are performed, and if the kernel is deemed untenable, evaluation ceases and the loop reverts to software execution.

First, Garpcc eliminates paths containing infeasible operations—those not supported by the backend synthesis tools. Since these prunes are mandatory, no estimates are required, and therefore this phase is performed before DFG construction.

Secondly, Garpcc prunes until the kernel fits into the available coprocessor resources. The current DFG is constructed at the start of this phase. Prunes are chosen in order of best (highest) ratio of area savings to additional exit count. After each prune, the DFG is updated and optimizations are re-applied.

Finally, additional prunes are applied iteratively for as long as one exists that increases overall performance. All costs are considered, including pipeline refilling etc. as described above. As before, the DFG is updated and re-optimized after each prune.

The final kernel's estimated performance is compared against the estimated performance of software (instruction processor) execution. For various reasons it may not be possible to form a kernel that benefits from using the coprocessor; in that case Garpcc reverts the entire loop to software execution.

5. Observations

In the optimization problems here, iterative approaches are suboptimal. Related work therefore considers exhaustive approaches, which evaluate all compatible groups of prunes [2].

The evaluations routines assume each kernel is loaded once and that thereafter it can be found in Garp's configuration cache, greatly reducing reconfiguration time. Simulations found this assumption to be false in many cases, indicating the need for a global culling of accelerated loops to reduce configuration thrashing as performed in [4].

Also, the kernel selection algorithm is very sensitive to profiling data; when actual runtime patterns differ greatly from the provided profiling data, performance using the coprocessor can in fact be much worse than straight software execution. Further research into more conservative kernel selection algorithms and/or dynamic (runtime) hardware/software choice is indicated.

In benchmarks, successful kernels have been observed with all 8 combinations of pruning from the 3 phases (feasible and/or fitting and/or performance). As expected, general purpose applications saw more need for fitting and performance prunes.

References

- [1] T. Callahan and J. Wawrzynek. Instruction-Level Parallelism for Reconfigurable Computing. In *Proc. FPL'98*.
- [2] T. J. Callahan. *Automatic Compilation of C for Hybrid Reconfigurable Architectures*. PhD thesis, University of California at Berkeley, Dec. 2002.
- [3] J. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proc. FCCM*, Napa, CA, Apr. 1997.
- [4] Y. Li, T. Callahan, E. Darnell, R. E. Harr, U. Kurkure, and J. Stockwood. Hardware-Software Co-Design of Embedded Reconfigurable Architectures. In *Proc. 37th DAC*. June 2000.
- [5] 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.