Coconut: *A Tool for Verifiable, High-Performance Image and Signal Processing*

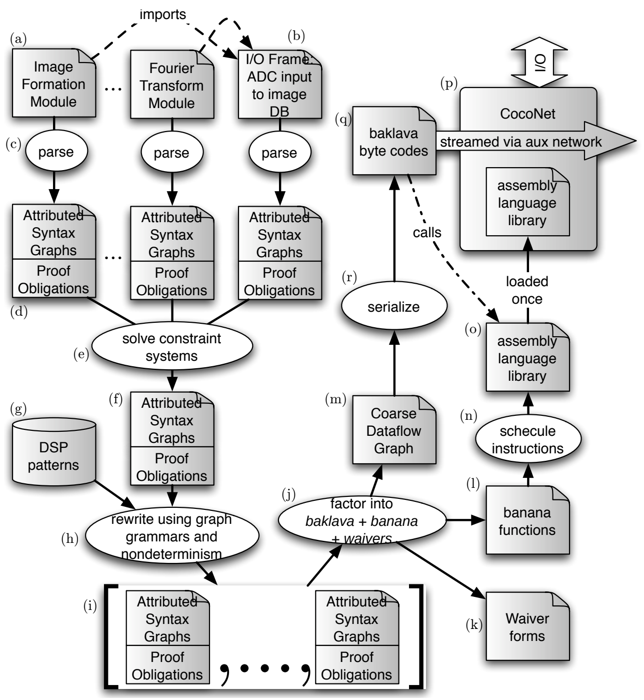
Christopher Kumar Anand, anandc@mcmaster.ca
Wolfgang Thaller, Gordon Uszkay,
Computing and Software, McMaster University

Abstract

Coconut is a tool for rapid development of safe, high performance signal processors, *e.g.* medical imagers. Domain specialists contribute via tailored formal specification languages. A high-level mathematical language provides an extensible type system and multi-deterministic expressions (joins) for defining relationships and attributes of physical systems. A declarative assembly language allows user input to algorithm and instruction selection. We have observed hundred-fold efficiency improvements from prototyped symbolic model transformations. Factoring parallel execution into a simple formal synchronization language plus atomic (serial) computations makes it easier to prove that the parallelization is correct.
Of most interest to this workshop, are the efficiency improvements which result from combining high-level problem structure with low-level device-specific information. This is expecially useful for architectures with rich instruction sets, but straightforward implementations giving control to the user. That is why we are targeting PowerPC + Altivec / VMX, and CELL SPU ISAs. Using our patterns for efficient inlined evaluation of special functions using Altivec/VMX, we have measured a 30 times improvement for sine/cosine pair (*i.e.* complex exponential) vector calculations over Apple-supplied libc. We are working with IBM Toronto to bring these improvements to SPU implementations of MASS, the vendor-supplied math library. For single precision, we expect an overall three-fold improvement over the existing optimized library.
We are starting to see the benefit of having a flexible, open compiler for doing rapid prototyping of programming language concepts and compiler experiments. For example, a *multiloop* is a novel control structure which leverges branch hints to turn upredictable nested control structures into optimal modulo schedules. We use advanced Magnetic Resonance spiral image reconstruction to illustrate the concept, the view to the user provided by our declarative assembly language, and the performance we get from our prototype modulo scheduler. Having an interpreter for SPU declarative assembler embedded in the functional programming language Haskell, enabled us not only to rapidly prototype the multiloop, but to develop a number of computational patterns, which can then be abstracted into inline functions or code transformations. These include the efficent evaluation of splines, special functions, outer products, and loop iteration without arithmetic, and a fused three-dimensional partial Fourier Transform. The mechanics of doing this in an interpretable functional language makes it easy to unit test not just functions, but code transformations.
Of course, new control constructs wouldn't be useful without corresponding instruction schedulers, and we're quite happy with our current scheduler which is within two percent of the theoretical lower bounds for the complicated 5K instruction MRI example.

Coconut Architecture

A Coconut program consists of a set of high level specifications organized into reusable modules (a), and a single I/O frame (b). These are written in HUSC, our mathematical specification language. These modules are parsed into fully typed abstract syntax graphs, and proof obligations for the module (d). The I/O frame specifies storage models and assertions for relations with hardware. Modules imported by the frame are merged by graph rewriting (*e.g.*, function inlining, dead code elimination). Parsing of a first definition of this language, together with type inferencing using a mixture of Constraint Handling Rules and Prologue has been implemented and is being tested.


imports
(a) Image Formation Module
Fourier Transform Module
(b) I/O Frame ADC input to image DB
(c) parse
parse
parse
Attributed Syntax Graphs
Proof Obligations
(d)
(e) solve constraint systems
(f) Attributed Syntax Graphs Proof Obligations
(g) DSP patterns
(h) rewrite using graph grammars and nondeterminism
(i) Attributed Syntax Graphs Proof Obligations
(j) factor into baklava + banana + waivers
(k) Waiver forms
(l) banana functions
(m) Coarse Dataflow Graph
(n) schecule instructions
(o) assembly language library
(r) serialize
(q) baklava byte codes
calls
loaded once
(p) CocoNet
streamed via aux network
assembly language library
I/O

At the next stage, (h), the ASGs must be rewritten in terms of ASGs containing only implementable edges (*i.e.* machine instructions and combinators encapsulating high-level control flow of representable data types). This rewriting is nondeterministic, and depends on a database of rewrite rules encompassing both DSP patterns and assembly-language fragments, (g). Currently, we are developing patterns directly in the lower-level declarative assembly language, and as ASG manipulation procedures written in Haskell, bypassing this important module. Finding the right level of abstraction is important, so we want a lot of examples before we start designing.
Although the rewriting is nondeterministic, resulting in multiple ASGs , we only show one ASG flowing out of this list of possibilities to (j) where the ASG is factored into coarse and fine-grained code graphs. This factorization is akin to the identification of basic blocks by conventional compilers, but in this case we want to identify large code chunks with serial execution, which can be expressed in a subset of the banana language (l), which consists of functions containing declarative assembly and a limited number of control-flow combinators, with known efficient implementations in assembly language. The multiloop, described below, is the first novel combinator. Identifying such control-flow patterns will grow

in importance over time as branch-prediction misses become more expensive, and as user-directed instruction fetch becomes more common.

All functions have a common, variable-argument binary interface with arguments passed on a "movable stack", optimized for remote procedure calls and command streams. These functions are scheduled (n) using a relatively expensive modulo scheduler into pipelined assembly code (o) in a simple binary module format, and loaded onto the target computer cluster (p) before program execution.

The coarse dataflow graph contains the scheduled assembly functions as special edges, collapsed from identified subgraphs in the previous factoring, together with a limited set of combinators representing concurrent dataflow (m). These graphs are serialized (r) into a stream of function calls, communication, synchronization and sequencing primitives called baklava, whose simple grammar and semantics make stream generation, manipulation, state-machine processing, and verification easier. These instructions are streamed through the run-time system, CocoNet (p). We are currently using rewriting logics to implement pre-run-time verification that the parallelization (which occurred in (h)), is correct, *i.e.*, that the parallel program produces a result (no deadlocks), which is independent of timing. In addition, all memory allocation is done at compile time (again in (h)), which allows pre-run-time verification of performance requirements and makes the run-time simple to verify by hand. We have a prototype parallelizer which annotates an abstract code graph for MRI reconstruction with resource assignments and generates baklava.

In addition to the compiled functions, and interpreted baklava stream, we will also factor out a complete set of proof obligations. This includes all of the assumptions about algorithm behaviour beyond the capability of the type inferencer to prove, *e.g.*, appropriateness of discrete approximations to continuous functions, and all input and output specifications. These obligations could be output as waivers of responsibility (k) for insuring that these statements are true.

Modulo Scheduling the Multiloop

Semantically, a multiloop is equivalent to a loop containing conditional execution. For performance reasons we implement it as a sequence of versioned loop bodies without conditional execution. In addition to the desired computation, each loop body calculates the next version of the loop body to execute. This transformation makes sense when the conditionals are not statically predictable, and are too expensive to predict at run time. In the case of the CELL SPU ISA, multiloops can be implemented using branch hints to reduce or eliminate delays caused by instruction fetching. Multiloops can be unrolled when the loop bodies are not sufficiently long for branch hinting to be effective. Unrolling by a factor of N increases the number of versions of the loop body from w to w^N, however, so must be done with caution. As with simple loops, modulo scheduling can be used to reduce code size, and increase efficiency. It is especially important for large loops with many versions. We are implementing a modulo scheduler with heuristics tuned to scheduling large multiloops. One way such structures arise is when a computation, which is the composition of several operations on an array or stream of elements, is implemented as a loop. In our case, the loop will be generated after manipulating the abstract mathematical specifications, but in conventional imperative languages, the loop is likely to be coded as a series of loops which are fused by the compiler.

Our test problem comes from next-generation MRI. Corrected spiral resampling is one part of a Non-Uniform Fourier Transform strategy in which samples of a function, $\mathbb{R}^3 \to \mathbb{C}$, at arbitrary points in $\mathbb{R}^3$ are convolved with a continuous kernel and accumulated at a regular grid of points, to produce the usual input for a 3D-FFT. Think of it as a loop fused out of parts: current position calculation, modulo reduction to determine alignment with the grid, sine/cosine calculation, complex multiplication, three by four 8-segment spline calculations to evaluate the kernel, an outer product ($4\times4\times4$), complex addition of outer-product with a backing store. A naive implementation would require 128 load/store operations/cycles per input value just to read and write the backing store. By caching the backing store in registers, and rotating it as necessary to reduce load/store overhead, and effectively using the SIMD capabilities to compute positional data, sine/cosine pairs and outer products efficiently, we can reduce the codegraph significantly.

By introducing different cases of the multiloop according to all allowed motions with respect to the backing-store, and incorporating alignement state with respect to the backing-store, we end up with a combined code graph of 5K instructions, 54 cases, and lower-bound cycle times of between 104 and 133 per iteration, determined by the number of SPU pipeline0 (arithmetic) operations. There are two shortest cases corresponding to non movement, which are also the hottest cases.

My student, Wolfgang Thaller, has made a lot of progress on a modulo scheduler for multiloops. The current scheduler implements several published modulo scheduling algorithms, and several new ideas needed to schedule multiloops. Ignoring register allocation–which given the 128 registers on a CELL SPU, we thought of as a lower priority–he has been able to schedule the entire test case with expected efficiency within two percent of the theoretical minimum. Running time for the scheduler is several hours on a single 2.5 GHz PowerPC 970, using the pure, lazy functional language Haskell, and C++ for the innermost search loop. Unfortunately, the current register allocation exceeds 128 architected registers for the test case, so he is implementing a more complicated register allocation.

Mixing Assembler and Functional Programming

Faced with the practical problem of trying to make good use of the rich CELL SPU ISA, I found it necessary to make frequent small experiments, to see how old design patterns adapt to a new ISA, and to try new ideas. Eventually, these design patterns will be captured as code transformations in Coconut, but first they must be discovered and tested. Having a declarative assembly language embedded in the lazy functional language, and the option of interpretation or code generation, has accelerated this discovery process, and makes the transition from writing specific code to writing generic code seamless. For example, moving loop induction calculations from arithmetic pipeline 0 operations to pipeline 1 permutations was developed in this way. The outer product mentioned above was actually developed using string permutations representing subexpressions, sorted lexicographically, and then translated into SPU instructions.