# Declarative Assembler

Christopher Kumar Anand[1], Jacques Carette[1], Wolfram Kahl[1], Cale Gibbard[2], and Ryan Lortie[1]

[1] McMaster University, 1280 Main St. West, Hamilton, Ontario Canada L8S 4K1
[2] University of Waterloo, Canada

**Abstract.** As part of a larger project, we have built a declarative assembly language. This language enables us to specify multiple code paths to compute particular quantities, giving the instruction scheduler more flexibility in balancing execution resources for superscalar execution. The instruction scheduler is also innovative in that it includes aggressive pipelining, and exhaustive (but lazy) search for optimal instruction schedules. We present some examples where our approach has produced very promising results.

## 1 Introduction

While developing a Domain Specific Language (DSL) for an MRI application (Coconut – see next section), we encountered a need for a DSL of a wholly different nature: we needed to specify choices amongst different "equivalent" computation paths made up of low-level assembler. An intelligent instruction scheduler will choose the best path, using built-in knowledge of the intricacies of a modern, vectorized and pipelined CPU architecture. Our collective experience told us that we should be specifying our problem in a declarative manner, to give maximal freedom to the scheduler. But we also knew that we could not schedule code that was not made up of specific assembler instructions. We decided to see if we could get these rather different paradigms (declarative and assembly) to coexist, and furthermore to serve as the main language for our compiler's back end. This paper reflects our success to date.

It is important to justify our claim, mainly that it is possible to design a new language which deserves the monicker of "declarative assembler". The key design points for our language were to

1. only describe dataflow
2. have built-in facilities for redundancies
3. have code that *looks like* assembler
4. control flow is, as much as possible, decided by the scheduler
5. be able to optimize (minimize) resource consumption

The first two criteria clearly indicate that a declarative language would be best; with that in mind, the last two criteria suggested to us that we replace the usual unification-driven language by an optimizing-scheduler driven language, but that these algorithms would otherwise fulfill the same operational criterion.

To fulfill the criterion that the code looks like assembler, we re-interpreted all non-branching instructions (of the PowerPC 745X and PowerPC 970) as "atomic" dataflow equations. This was done in part to isolate at the pure dataflow subset of the language from the control part, mainly because branching instructions on these platforms are so much more expensive, and look to get even more expensive on future architectures. For the target applications, we know from experience that a lot of control flow can be eliminated by clever use of permute and select (logical) instructions. We made the decision to eliminate branches from the declarative assembly language, and express all control flow which cannot be eliminated by use of permutation and selection in 'combinators'. So the conflict between "declarative" and "assembler" is solved by eliminating all of the problematic instructions.

The only dataflow equations that we would allow would be those that were given by actual assembly instructions. In other words, instruction selection would already be done by this stage of the compiler. However, we still have to contend with criterion 2: there would be alternatives paths to compute equivalent answers[3] offered to the scheduler.

We like to draw an analogy with the cut-free subset of Prolog: where Prolog specifies facts and rules, we give dataflow equations, and instead of using unification for realizing solutions for a query, we use a scheduler to jointly realize an input-output relation and optimize its resource usage. Crucially, the fact that we are optimizing instead of solving, we can use redundant information in dataflow equations. However, the issue of register allocation does complicate matters greatly.

Figure 1 shows a code graph that arises in an implementation of a non-uniform Fourier transform. The nodes are labelled with machine instructions (with immediate constants) or a constant value. Arrows are in the direction of data flow and the nodes are shaded to indicate the execution unit on the PowerPC 7455. The most important aspect of this example is the triple join, which allows us to produce a much better

[3] Our application often requires answers with, *e.g.*, 16 bits of accuracy, and we could have two different algorithms to compute this answer, one to 18 bits and the other to 20 bits of accuracy. We would let the scheduler decide which is best.

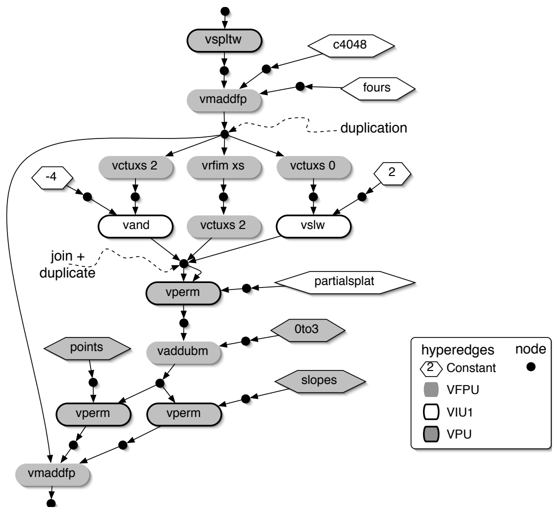

**Fig. 1.** Code graph for the calculation of uniform linear spline in two dimensions

schedule (depending on the context of the code fragment) than current compilers can achieve.

We were quite pleased to find that even on relatively short segments of code, we can already achieve significant accelerations. In our second example, we show that a near-optimal schedule of our implementation of map $\theta \mapsto e^{i\theta}$ is 31 times faster than standard library calls. Even if we do a “fairer” comparison by giving `gcc` the same code and enable processor-specific optimizations, we are still 2.5 times faster.

The next section gives a quick overview of Coconut, including the application domain and our approach to the problem. This serves as motivation for the rest of the paper, which concentrates on the declarative assembly language. We first describe our main semantic tool, namely code graphs. We next describe the scheduler, which gives our language its operational semantics; this section also includes additional details on the above two examples, as well as explicit timing results. We review some related work before drawing some general conclusions.

## 2 Coconut

Coconut addresses the needs of high-performance signal processing applications like diagnostic medical imaging. These applications involve complex physical and mathematical models and tight performance requirements. Engineers are forced to make low-level code optimizations which depend on high-level model characteristics which are likely to change. Engineers and scientists using Coconut will use DSLs tuned to their level of abstraction, and Coconut will use whole-program compilation to synthesize these contributions into optimal parallel, vector code. At the lowest level, engineers and programmers (sometimes called “DSP Gurus”), will apply optimization rules to existing code, or translate MATLAB prototypes to efficient C code. They will write rules for implementing high-level operations, and transformations for combining high-level operations. Declarative assembler is the language they will use to encode the most basic building blocks: efficient implementations for atomic operations, such as matrix multiplication and trigonometric function evaluation.

The novel features of our declarative assembly language correspond to two techniques used by hand-coders: clever instruction sequences implementing operations common in the application domain, (easily expressed in declarative assembly), and code transformations for implementing common higher-order functions.

Although by separating high-level specifications from architecture-specific implementations, we will be able to (relatively) rapidly retarget our code generation, we will only implement code generation for one Instruction Set Architecture (PPC+Altivec) and two implementations (IBM 970 and Motorola 7455). We chose PowerPC+Altivec because PowerPC has dominated the market for high-end embedded signal processing in the recent past, and because, having a richer set of instructions, Altivec provides the greatest scope for agressive optimization. Also, the 970 and 7455 come from different generations of microprocessor design and present different opportunities for optimization.

## 3 Code Graph Syntax and Semantics

Our *code graphs* are a kind of directed hypergraphs that can be seen as generalisation of the *jungles* of [6], which are essentially a hypergraph presentation of term graphs with variables.

In this section, we first motivate the definition of our code graphs in detail; then we present the macro assembly language that is used as input language of our declarative assembler, and finally explain the translation from the macro assembly language into code graphs.

### 3.1 Code Graph Syntax

Term graphs are usually represented by graphs where nodes are labelled with function sysmbols and edges connect function calls with their arguments. An alternative representation was introduced with the name of *jungle* by Hoffmann and Plump [6] for the purpose of efficient implementation of term rewriting systems (it is called "term graph" in [10]).

A jungle is a directed hypergraph where nodes are only labelled with type information (if applicable), function names are hyperedge labels, each hyperedge has a sequence of input tentacles and exactly one output tentacle, and for each node, there is at most one hyperedge that has its output tentacle incident with that node (respectively *exactly* one such hyperedge in the absence of variables).

In our code graphs, hyperedge labels are either opcodes or constants, as can be seen in Fig. 1, where nodes are drawn with type information omitted, output tentacles are arrows from hyperedges to nodes, and input tentacles are arrows from nodes to hyperedges — the ordering relation between in- resp. output tentacles incident with the same hyperedge is not made explicit in the drawing, but is part of the graph structure. In comparison with jungles, our code graphs introduce two generalisations:

– Hyperedges can have a multiple output tentacles – this is necessary for opcodes that produce more than one result.[4]
– Several output tentacles may be incident with any one node — we call such tentacles *joining*.

Since *joins* are a novel feature in assembly languages, we need to justify their inclusion. In short, they are used for results that can be obtained in different ways, and also for situations where different intermediate values could be used interchangeably. The following are typical applications:

– **Multiple entry points:** Many common math functions are implemented (for the sake of efficiency) via algorithms with extra preconditions, and initial code ensures that those preconditions are satisfied. For example,
  - trigonometric functions are only calculated on a fundamental domain, and modulo calculations are first performed to put arguments into the fundamental domain;
  - some functions have a standard interface (e.g. choice of units), but an alternative interface is much more efficient, so the initial statements perform the necessary conversions.

  In both cases, we can eliminate the respective initial instructions if we can verify the stricter preconditions, or if we can rewrite the upstream calculation to produce results matched to the more efficient interface.

[4] In all known examples, the second result is always a condition code, i.e. carry or overflow, but we think it will be better to treat all results uniformly.

- **Instruction selection,** For example merging disjoint bitfields by either logical *or* or arithmetic *add* instructions uses different functional units on some processors. In such cases, conventional optimizing compilers switch instructions to get better schedules; in our approach, we emit a join in the assembly code and let the code graph scheduler select the better branch in each case.
- **Multiple code paths** beyond single-instruction alternatives: It is possible to do some computations in different units, e.g., evaluating polynomials in the scalar floating point unit or the vector floating point unit. Such alternative code paths can be used in two ways:
  - `map f`, where `f` is a simple function, can be unrolled and performed simultaneously on different data in different execution units;
  - the code can be in-lined in different contexts where relative demand on execution units, register pressure, etc., vary enough to make one code path more efficient than another.

## 3.2 Code Graph Semantics

It is an established result that conventional term graphs, extended with sequencing information for source (input) and sink (output) nodes, are obtained as arrows of free gs-monoidal categories over conventional signatures [3]. GS-monoidal categories can be obtained as a generalisation of cartesian categories[5], where the naturality conditions are weakened for the transformations of duplication (sharing) $\nabla_\alpha : \alpha \to \alpha \times \alpha$ and termination $!_\alpha : \alpha \to \mathbb{1}$, where $\mathbb{1}$ denotes the unit of $\times$, which in concrete set-based semantics is just any one-element set.

For accommodating the fact that hyperedges can have a multiple output tentacles it is suficient to replace conventional signatures with signatures where function symbols may have a vector of result types. For joins, we add an additional transformation $\varDelta_\alpha : \alpha \times \alpha \to \alpha$, together with appropriate laws. In addition, we need co-termination $¡_\alpha : \mathbb{1} \to \alpha$ for garbage collection of operations that have no inputs supplied — termination ! is used for garbage-collection of unused results.

The part of the graph in Fig. 1 from the input up to and including the first permutation (and ignoring the long edge) will then be translated into the following expression:

[5] The basic language of strict monoidal categories has identities $\mathbb{I}_\alpha : \alpha \to \alpha$, sequential composition "$;$" such that $(f;g) : \alpha \to \gamma$ for $f : \alpha \to \beta$ and $g : \beta \to \gamma$, and parallel composition $\otimes$ such that $(f \otimes g) : (\alpha \times \beta) \to (\gamma \times \delta)$ for $f : \alpha \to \gamma$ and $g : \beta \to \delta$.

$$\begin{gathered}(\mathsf{vspltw} \otimes \mathsf{o4048} \otimes \mathsf{fours}) ; \mathsf{vmaddfp} ; \nabla ; (\nabla \otimes \mathbb{I}) ; \\ (((\mathsf{neg4} \otimes \mathsf{vctucs\_2}) ; \mathsf{vand}) \otimes (\mathsf{vrfirm\_xs} ; \mathsf{vctucs\_2}) \otimes ((\mathsf{vctucs\_0} \otimes 2) ; \mathsf{vslw})) \; ; \\ (\mathit{\Delta} \otimes \mathbb{I}) ; \mathit{\Delta} ; (\nabla \otimes \mathsf{partialsplat}) ; \mathsf{vperm}\end{gathered}$$

The first line extends up to the upper duplication node and its three outgoing tentacles, and the second line ends just before the three joining tentacles.

Formalising our code graphs in a simple extension of gs-monoidal categories has the advantage that this provide us with a combinator language for our code graphs that easily accommodates the macro mechanism, and the relational (multi-algebra) semantics of [4, 5] also carries over without problems ($\otimes$ is then parallel composition of relations):

- co-termination is interpreted as the empty relation, and
- joining $\mathit{\Delta}_\alpha$ is interpretated as the relation $[\![\, \mathit{\Delta}_\alpha \,]\!]$ with

$$(x, y)[\![\, \mathit{\Delta}_\alpha \,]\!]z \qquad \Leftrightarrow \qquad x = z \vee y = z \;\; ;$$

This way, if the semantics of each opcode can be given as a relation (typically a function) from its inputs to its outputs, then each code graph $G$ with types $\langle \alpha_1, \ldots, \alpha_m \rangle$ of input nodes and types $\langle \beta_1, \ldots, \beta_n \rangle$ of output nodes has as semantics a relation between the corresponding cartesian products:

$$[\![\, G \,]\!] : \alpha_1 \times \cdots \times \alpha_m \leftrightarrow \beta_1 \times \cdots \times \beta_n$$

This relation is interpreted as the set of total functions contained in it; the scheduler is permitted to implement any of these functions, and essentially does this by choosing one alternative from each join.

With this semantics, proving that a code graph $G$ satisfies a specification entails showing that each each total function contained in $[\![\, G \,]\!]$ is correct with respect to the specification under consideration. When the specification is also given as a relation (see [7]), then this correctness condition boils down to a simple inclusion of relations.

Equipped with this understanding, several improvements fit in without any problems. Amongst these are "non-deterministic constants": Constants are represented by hyperedges with typically one output tentacle and no input tentacles. Such hyperedges can be labelled with expressions denoting a *set* of possible values.

### 3.3 Macro Assembly Language for Data-Flow Code Graphs

In concrete syntax, a code graph of the Coconut Assembler language is presented in a way that is very similar to the equation systems used by Ariola and Klop [2] to denote (cyclic) term graphs. This equational approach allows us to write down sub-trees of a code graph in conventional expression notation, and expresses sharing (and, in the case of [2], cycles) through references to node names.

In our language, the following additional features are present:

- Left-hand sides can be either single identifiers, or pairs of identifiers. The latter will be used for opcodes that return two results.
- Joins are represented by identifiers that occur on several left-hand sides.
- Macro definitions encapsulate graphs with an arbitrary number of *interface nodes*; macro instantiations can be mixed with equations in graph definitions. A macro instantiation is understood to connect a copy the encapsulated macro graph via the interface node instantiations with nodes of the containing graph. Macro instantiations are statements instead of as expressions since this adds flexibility to potential reuse of macro components. All identifiers occurring in a macro definition, including in its interface, are *local*.
- Statements can, besides equations, macro instantiations, and macro definitions, also be just node declarations; this enables type declarations to be attached to nodes that otherwise occur only in macro instantiations.

Complete assembly programs are passed to the scheduler as macro instantiations of the shape $\mathsf{foo}(\mathsf{in}_1, \ldots, \mathsf{in}_m, \mathsf{out}_1, \ldots, \mathsf{out}_n)$ in an environment where all necessary macro definitions are visible.
The resulting grammar has the following productions:

| | |
|---|---|
| *node* | $\rightarrow \mathit{identifier}\ \texttt{::}\ \mathit{type} \mid \mathit{identifier}$ |
| *expr* | $\rightarrow \mathit{identifier} \mid$ **numeric constant** |
| | $\mid \mathit{opcode}\ \mathit{expr}\texttt{,}\ \ldots\ \texttt{,}\ \mathit{expr}$ |
| *statement* | $\rightarrow \mathit{equation} \mid \mathit{node} \mid \mathit{macro\text{-}instantiation}$ |
| *macro-definition* | $\rightarrow \mathit{identifier}\ \texttt{(}\ \mathit{identifier}^{+}\ \texttt{)}\ \{\ \mathit{statement}^{*}\ \}$ |
| *macro-instantiation* | $\rightarrow \mathit{identifier}\ \texttt{(}\ \mathit{identifier}\ \texttt{=}\ \mathit{expr}\texttt{,}\ \ldots\ \texttt{)}$ |
| *equation* | $\rightarrow \mathit{node}\ \texttt{=}\ \mathit{expr} \mid \texttt{(}\mathit{node}\texttt{,}\ \mathit{node}\texttt{)}\ \texttt{=}\ \mathit{expr}$ |

Type annotations on nodes can be used for documentation and are checked; type checking mainly relies on the type information available for opcodes.

Type checking uses the typing of assembly instructions only at a very low level, essentially only to distinguish the the kinds of registers that can be assigned to nodes by the scheduler. The rationale behind this is that apparently inconsistent use of the same value at different types may still be correct with respect to a (typically relational, non-univalent) specification. In fact, in one of our examples dual use of the same numeric constant both as a (redundant) bit mask and as a floating point value saved a register and thus enabled a tighter schedule.

## 3.4 Assembly into Code Graphs

Macro assembly programs are first converted into *direct code graph equation systems* by recursively performing the following steps:

- For each expression $e$ that is not just an identifier, for each occurrence of $e$ as an argument of either an opcode or a macro instantiation, a new node identifier $n$ is generated, the occurrence is replaced by a reference to $n$, and an equation $n = e$ is inserted.
- Macro instantiations are expanded by
  - for each argument assignment $n = e$ in the argument list of the macro instantiation, deriving an instance of the macro body by replacing each occurrence of $n$ with $e$, and deleting all equations that would have a non-identifier on the left-hand side.
  - obtaining the direct code graph equation systems from that instance of the body, and
  - including the resulting equations and declarations into the calling scopem after renaming all clashing identifiers.

A direct code graph equation system defines a code graph as follows:

- Numeric constants only occur as right-hand sides of equations; each numeric constant corresponds to a no-input-single-output hyperedge incident with the node on the left-hand side of its equation.
- Opcodes only occur top-level on right-hand sides, and have only identifiers as arguments. Each equation with an opcode on the right-hand side corresponds to a hyper-edge with output tentacles incident with the nodes corresponding to the equation's left-hand side, and with input tentacles incident with the nodes corresponding to the opcode arguments.

In the process, some nodes may receive several output edges since their identifiers occurred on the left-hand sides of several equations — these are the join nodes.

The top-most call induces a directed environment, from $\mathsf{in}_1, \ldots, \mathsf{in}_m$ to $\mathsf{out}_1, \ldots, \mathsf{out}_n$, and we currently only consider cycle-free graphs, so we can perform garbage collection after each macro expansion: Operations whose results are never used, and operations that do not have all necessary inputs supplied to them can be deleted – normally the latter will delete unused alternatives in joins. If this results in deletion of any $\mathsf{out}_i$, this is of course an error.

## 4 The Pipeline Scheduler

### 4.1 Declarative Aspects of the Scheduler

Scheduling a pure data flow is equivalent to solving a constraint problem, where the constraints arise from data dependency, latency of instructions, finite number of physical and logical registers, limits on the number and types of instructions which can be dispatched per cycle, other constraints whose form varies from processor to processor (including size and organization of dispatch and completion queues).

We solve the problem using a greedy algorithm with backtracking, implemented in Haskell using lazy list generation and pruning. Given our target applications, we set two somewhat contradictory aims: given a maximum number of cycles, exhaustively list the feasible schedules; find at least one such "quickly".

The basic algorithm is to start with outputs and put them in a list of possible instructions, and start scheduling the last machine cycle, working backwards in time. At each step we try all possibilities of scheduling instructions, and we follow the input tentacles of these instructions. If we encounter a join, we try adding each of the hyperedges in turn to the schedule set. After scheduling, we solve a second constraint system to assign physical registers to the hyperedges which represent instructions, as well as loading the constants into registers. Some feasible schedules may not have feasible register allocations.

The execution time of the basic algorithm would be exponential in the code graph size, so we must prune partial schedules which we know cannot be completed to feasible schedules. The pruning code is the main source of complexity in the scheduler.

**Example:** In Figure 1, we show the code graph for a fragment needed for efficiently computing nonuniform Fourier Transforms as described in [1]. This operation requires the evaluation of a function of two variables defined as the tensor product of two uniform linear splines, on a $4x4$ regular grid of points. This fragment evaluates the uniform linear spline on 4 points with uniform separations known in advance. It occurs twice in parallel in the complete routine. Within each of the spline calculations, there is a three-way join, indicating multiple equivalent code paths. In this example, the alternative code paths are each of the same length, with similar latencies, but on the target processors one of the paths requires the same (vector floating-point (VFPU)) execution unit for both instructions, while the other paths use different units (the VFPU and vector simple integer unit). In the common case that the vector-floating-point unit is the bottleneck, one of the other two execution paths will always be picked, but if the complete code balances the execution unit usage, it may be more efficient to use different alternative code paths in the left and right instances. Which of these execution paths is more efficient may depend on which constants are used elsewhere in the code graph, because one path requires a register to be loaded with the value 2 and the other with the value $-4$. If either of these constants can be shared, register pressure may be reduced, which could result in a more efficient overall schedule.

### 4.2 Adding Control Flow

Of course, we are often interested in scheduling a program which does have control flow, for example a program which maps a function over an array of values. Wrapping a schedule for the declarative function in a loop would do this, but is unlikely to be maximally efficient. While we could unroll the loop and gain performance in this manner, we would also increase code size. An even better option is to pipeline the loop body, so that in iteration $n$ we are in the process of doing part of calculations $n, n+1, ..., n+k$, where $k$ is the number of stages. This is the only control flow pattern we currently recognize. Scheduling pipelined loop bodies is the same as scheduling pure dataflow, except that we have to add extra constraints for conflicts between different stages.

### 4.3 Example: Map of $e^{i\theta}$ over a floating-point vector

In this example we read in a vector of $\theta$ values and calculate vectors of $sin(\theta)$ and $\cos(\theta)$. We read the vector in 128 bit short vector chunks, and process the four 32 bit floats in parallel. The dual dependency graph for this is shown in Figure 2, with constants omitted, and each hyperedge drawn as a node in the column corresponding to its stage of execution, and the row corresponding to the cycle in which it is dispatched. Arrows

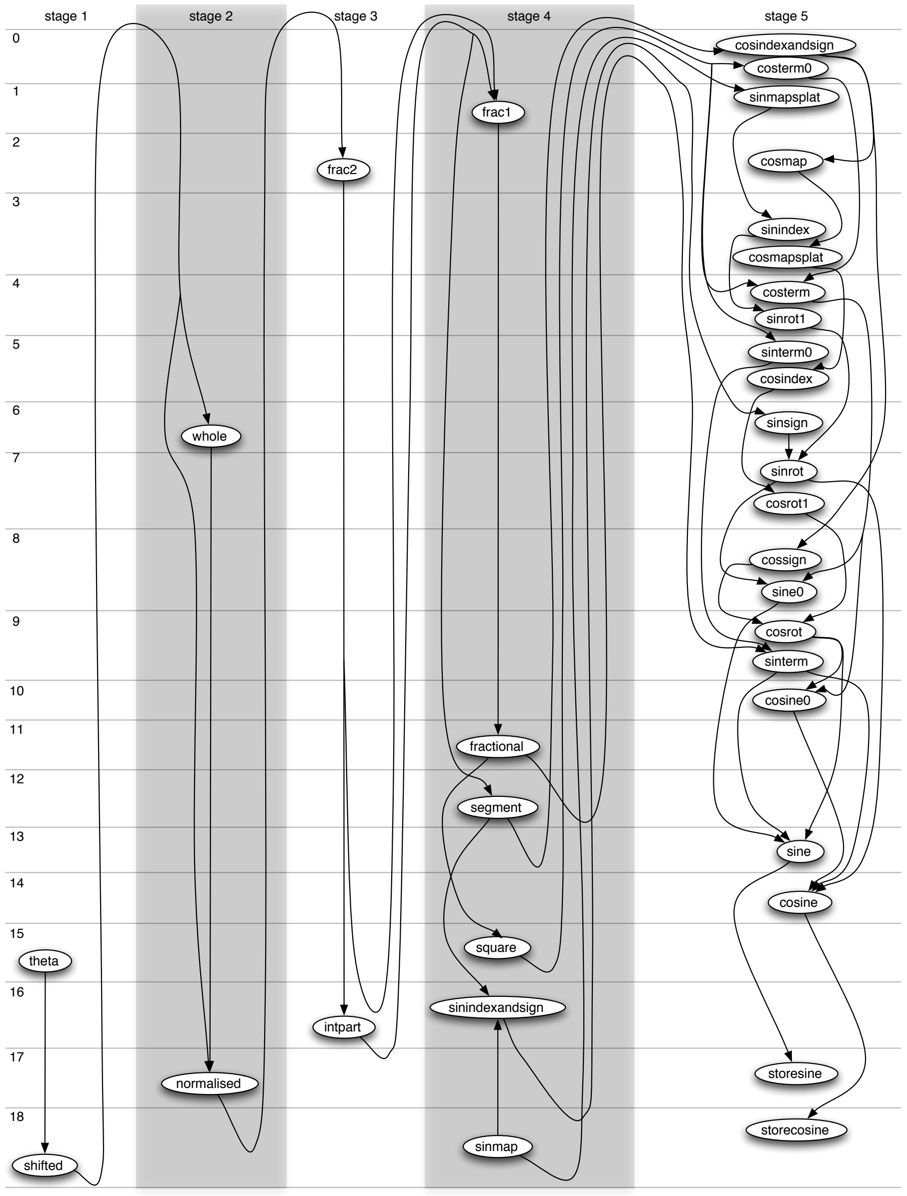

**Fig. 2.** Dual dependency graph for the calculation of sine and cosine pairs

extending from one column to the next are carried forward to the next stage. Note that this code graph is entirely declarative–with no control flow. Most instructions are not dispatched in consecutive cycles, because of execution latency constraints. This scheduled loop body executes in 19 cycles, which are separated in the diagram by horizontal grey lines, and are labeled by the numbers 0 to 18. The code graph is scheduled over 5 stages, indicated by shaded and unshaded columns, which means that the value of $\theta_n$ loaded in cycle 15 of iteration $n$ of the loop will be processed during iterations $n$ to $n+4$, and at the end of iteration $n+4$, $\sin(\theta_n)$ and $\cos(\theta_n)$ will be stored. To be useful, loop mechanics (a branch and incrementing of the induction variable and possible pointer arithmetic) must be added, but our scheduler finds schedules to which these instructions can be added at zero cost.

| | cycles/float | versus libc | versus Coconut |
|---|---|---|---|
| libc | 74 | - | 31 |
| vecLib | 26.65 | 2.8 | 11 |
| gcc | 5.9 | 13 | 2.5 |
| Coconut | 2.4 | 31 | - |

**Fig. 3.** Performance of $\theta \mapsto e^{i\theta}$ applied to an array in L1 cache.

Figure 3 gives the performance results of running this code on a floating-point array, with the results being stored to separate floating-point arrays. The *libc* code is a C loop containing calls to the standard math library (double precision, since it was faster). The *vecLib* code uses short vector calls from the accelerate framework installed with Mac OS X 10.3. The *gcc* code is the code graph from Figure 2, written in C using the Altivec extensions built into gcc. The first three examples were compiled with `cc -O3 -mcpu=7450`, to enable processor-specific scheduling. The Coconut code uses the same code graph, written in our declarative assembly language and scheduled with our current prototype compiler.

Figure 2 shows our schedule using 5 stages and 19 cycles, or 2.375 cycles per floating point operation, which is very close to what we measure. Our current scheduler only partially enumerates valid register allocations, so there may be a shorter schedule. However this code contains 17 vector-floating point instructions, only one of each can be dispatched per cycle, thus there is no schedule with fewer than 17 cycles.

## 5 Related Work

Somewhat to our amazement, a literature seach did not turn up any previous effort at designing an assembler-level declarative language. There is clearly some related work in the low-level code optimization literature (which is too vast to cite), but a fundamentally different approach is taken there: existing, or recently generated, assembler code is taken as a specification, passed through advanced and clever code analysis, from which some candidate semantics-preserving code transformations are derived and applied to find more efficient code sequences. To us, this resemble a typical local optimization technique[6], while our approach is in some sense global, at least over a given code chunk. We feel that this is why we are able to obtain a better schedule in the linear spline example.

Others have already given assembly language a more modern treatment. Of these, the work on Typed Assembly Language (TAL) [8] and the "higher-level" assembler of `C--` [9] stand out. Like TAL, our assembler is typed, but as we strive for extreme efficiency, we have had to abandon portability altogether.

It would of course be interesting to see if our ideas can be generalized to other assembly languages than the PowerPC. The thorough information contained in [11] about the design of machine-description languages is an excellent starting point. The experience gathered with the SALTO system [12] is also quite relevant.

## 6 Conclusion and Future Work

We are pleased that our initial idea of combining a declarative paradigm with a low-level assembly language was not only realizable, but in fact gives results which significantly outperform conventional compiler technology. Certainly our language design has brought a tremendous amount of clarity to our own internal discussions on instruction selection and on the development of our prototype scheduler. However, this is just a small step in the much larger design and implementation of Coconut.

We expect to enrich our assembly language significantly in the next few months. Our first order of business will be to investigate the expressive power of combinators on code graphs. We hope to be able to use these combinators to express larger patterns instead of control-flow based code fragments. The use of combinators should allow us to continue to

[6] where optimization here is taken in the sense of smooth optimization of an objective function

do very aggressive instruction scheduling and pipelining, by making important semantic information available to the scheduler, thus obtaining even larger performance gains on medium-sized code chunks. Currently we write different schedulers for each combinator, because we don't have enough experience and examples to know how to abstract this into a scheduler-combinator language to parallel the code graph combinators.

## References

1. C. K. Anand, T. Terlaky, and B. Wang. Rapid, embeddable design method for spiral magnetic resonance image reconstruction resampling kernels. *Optimization and Engineering*, 5(4):485–502, 2004.
2. Zena M. Ariola and Jan Willem Klop. Cyclic lambda graph rewriting. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 416–425, Paris, France, 4–7 '`jul`' 1994. IEEE Computer Society Press.
3. A. Corradini and F. Gadducci. An algebraic presentation of term graphs, via gs-monoidal categories. *Applied Categorical Structures*, 1999.
4. Andrea Corradini and Fabio Gadducci. Functorial semantics for multi-algebras. In J. L. Fiadeiro, editor, *Recent Trends in Algebraic Development Techniques*, volume 1589 of *LNCS*, pages 78–90. Springer, 1999.
5. Andrea Corradini, Fabio Gadducci, and Wolfram Kahl. Term Graph Syntax for Multi-Algebras. Technical Report TR-00-04, Dipartimento di Informatica, Università di Pisa, 2000.
6. B. Hoffmann and D. Plump. Jungle evaluation for efficient term rewriting. In J. Gabrowski, P. Lescanne, and Wolfgang Wechler, editors, *International Workshop on Algebraic and Logic Programming*, volume 49 of *Mathematical Research*, pages 191–203. Akademie-Verlag, 1988.
7. Wolfram Kahl. Refinement and development of programs from relational specifications. *Electronic Notes in Computer Science*, 44(3):4.1–4.43, 2003.
8. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):527–568, 1999.
9. Simon Peyton Jones, Norman Ramsey, and Fermin Reig. `C--`: A portable assembly language that supports garbage collection. In Gopalan Nadathur, editor, *Principles and Practice of Declarative Programming, PPDP 1999, paris, France*, volume 1702 of *LNCS*, pages 1–28. Springer, '`OCT`' 1999. Invited talk.
10. Detlef Plump. Term graph rewriting. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools*, chapter 1, pages 3–61. World Scientific, Singapore, 1999.
11. Norman Ramsey, Jack W. Davidson, and Mary F. Fernández. Design principles for machine-description languages. http://www.eecs.harvard.edu/~nr/pubs/desprin.pdf, 2000.
12. Erven Rohou, Francois Bodin, Andre Seznec, Gwendal Le Fol, Francois Charot, and Frederic Raimbault. SALTO: System for assembly-language transformation and optimization. Technical Report RR-2980, INRIA, 1996.