Automated Optimization of a Computation for Different Architectures

Ted J. Biggerstaff
Software Generators, LLC
Austin, Texas USA
dslgen at softwaregenerators dot com

Abstract—**DSLGen™ (Domain Specific Language Generator) is a program generation system in which application programs can be written in a domain specific language that is independent of the execution platform architecture and yet can be targeted to arbitrary existing and future execution platforms in a way that exploits the performance or computation improvement opportunities specific to those platforms. This allows switching from one execution platform to another without reprogramming the applications. The generation of target programs is fully automatic and requires no user input or action beyond the specification of the computation and the separate specification of the desired features of the target execution platform.**

Keywords -- associative programming constraints, natural and synthetic partitions, design patterns, logical and physical architectures, design feature encapsulation, implementation neutral specification, domain specific languages, inference, problem domain inference, partial evaluation.

I. INTRODUCTION

DSLGen™ (patents issued [8] [9] [10] and patent pending) is a transformation-based program generation system that fully automatically generates a target implementation optimized for a variety of target execution architectures from two independent specifications: 1) a domain specific, *Implementation Neutral Specification* (*INS*) of the desired computation and 2) a domain specific *EXecution Platform Specification* (*EXPS*) that describes what optimization features of the execution platform should be exploited. **The INS is *invariant* over all target execution platform architectures**. That is, an application programmer can make no predictions about the architecture of the target implementation by looking at the INS alone. Thus, no reprogramming of the INS is required to switch from one platform to another. Only the EXPS features need to be changed to switch from one architecture (e.g., multicore) to another (e.g., vector machines). Importantly, DSLGen™ fully automatically converts an INS and an EXPS into target implementation code that takes advantage of a broad range of opportunities for high capability computations including large grain parallelism (e.g., multicore CPUs), small grain parallelism (e.g., instruction level parallelism or ILP), design pattern frameworks and so forth. It is theoretically possible to extend DSLGen™'s capabilities to other target execution platforms such as GPUs, Digital Signal Processors (DSPs), specialized processors, Field Programmable Gate Arrays (FPGAs), and API interfaces to layered implementations or libraries. The author believes that DSLGen™ can be extended with new transform sets that will produce output optimized for virtually any arbitrary existing or future architecture. How can DSLGen™ automatically produce programs that are tailored to such highly varied execution architectures?

The short answer is that DSLGen™ is an extensible generator that is designed to create a program design from scratch based on the INS plus generalized constraints and design features specified in the EXPS. In some sense, it is doing what a human programmer does. DSLGen™ automatically builds a Logical Architecture (LA) that constrains some problem domain oriented features of the target program design but defers building a Physical Architecture (PA) that commits to programming language and implementation platform oriented features (e.g., routine architectures, parametric connections, communication patterns and synchronization patterns). That is, DSLGen™ architects, designs, constrains, reorganizes and optimizes the target program in the problem and programming process domains rather than in the programming language (PL) domain and only after the macroscopic structure of the program is settled does it generate PL level code. In short, it designs the solution first and codes it second.

Part of the secret to this process is that DSLGen™ eschews PL based representations during the design and architecture portion of the process thereby freeing it from the highly restrictive constraints of PLs. PLs are solution oriented not problem oriented. They require the programmer to tell how to perform a computation whereas during these early phases, the programmer knows what needs to be done and what design features the solution will have (i.e., the computational goals) but has not yet fully determined how to implement and integrate the computational needs and solution features.

II. THE PROBLEM

A key problem in exploiting the capabilities of various existing and future execution platform architectures for a specific target computation is the conflict between the goal of precisely describing the implementation of a target computation and the goal of casting the implementation into a diversity of forms each of which exploits a different set of high capability features of some specific execution platform architecture (e.g., parallel processing via multicore based threads). The key culprit in this conflict is the representation system used in the course of specifying the function of a target program – that is, the use of *programming language based abstractions* to represent the evolving program at each stage of its development and evolution. Einstein said "We see what our languages allow us to see." And when a computer scientist understands his or her world in terms of

programming languages, it is natural to construct intermediate design and precursor representations of programs in terms of programming language (PL) based abstractions. This has led to the conventional, reductionist or layered models of program designs.

These models may range from relatively concrete models built from minimally abstracted PL structures (e.g., abstract data types and object oriented models) to quite abstract models (e.g., Model Driven Engineering models [23]) that attempt to defer a greater number of concrete commitments in an effort to allow a greater diversity of eventual concrete program manifestations. The layers within the models may represent a variety of features or structures that eventually will be projected fairly faithfully into the structures of the final program (e.g., class diagrams). The layers also may represent specifications of behaviors (e.g., state diagrams) that will affect the final structure of the program in less direct ways but nevertheless are still partially expressed in terms of, and therefore imply, some structural elements of the final target program. The author believes that such PL based models have been a key impediment to mapping an implementation neutral specification of a computation to an arbitrary platform while still exploiting whatever high capability features that platform possesses. The problem, in summary, with PL based representations (regardless of their level of abstraction) is that the layers within the overall model of a computation have unintended and hidden interdependencies that arise because of the PL based abstractions. These often force implementation driven design goals (e.g., exploit multicore parallelism) to propagate complex, interrelated revisions and restructurings globally across many or all of the layers of the model. Why does this occur?

Typically in such a layered model, the structure and some details of a layer of the target program are specified along with some abstract representation of the constituent elements (i.e., lower level layers) of that layer. In human based application of layered design, the abstract elements of the lower level layers are often expressed in terms of a semi-formal pseudo-code or structural specification. In the automated versions of layered design, the informal specification is often replaced by somewhat more formal expressions (i.e., UML specifications [31]) of the interfaces to the lower level layers. For example, these might be simple routine calls, object oriented invocations, sub-types, sub-classes or skeletal forms of elements that remain to be defined. Alternatively, these interfaces may be calls to or invocations of concretely defined API layers or interfaces to message based protocols (e.g., finite state machine specifications). In any case, the structure is typically fixed at a high level before the implications of that structure become manifest in a lower level later in the development process. Refinements within the lower layers often require changing or revising the structure at a higher level, which can be problematic. Further, in an automated system, distinct programming design goals will be, by necessity, handled at different times. This is further complicated by the fact that multiple design goals may be inconsistent (at some level of detail) or at least, they may be difficult to harmonize.

A good example of this kind of difficulty is trying to design a program to exploit thread based parallel implementation. The exact structure and details of the final program are subtly affected by a myriad of possible problem features and programming goals. A threaded implementation will require some thread synchronization logic which may be spread across a number of yet to be defined routines. The computation will have to be partitioned into parts that are largely determined by the specifics of the target computation. These partitions will be mapped into routines and threads (e.g., some lightweight computations batched in one thread and other heavyweight computations decomposed into slices with their own threads). The thread protocol will introduce low level implementation details that potentially will have to be harmonized across a number of routines. The parameter choices for these routines (i.e., the plumbing) may be involved in the communication design for these thread routines and will be constrained by low level implementation details of the thread protocol. In DSLGen™, such programming language level routine structures, routine inter-communication decisions, thread protocol restrictions and thread library implementation requirements are added into the architecture close to the end of the design process after simplified logical architectures (LAs) of the elements of the domain specific computation specification (e.g., the INS) have been sketched out in broad general terms, terms that elide and defer much of the PL level detail. For example, the early LA design for thread based designs, vector machine designs, GPUs or others are indistinguishable from one another. Furthermore, the division of the computation into functions or routines and the parametric interconnection of those routines has yet to be decided when the LA is first sketched out.

If an automated generator tries to handle all of these various design issues at once, there is an overwhelming explosion of cases to deal with and the approach quickly becomes infeasible.

III. THE SOLUTION

The ideal solution would be to recognize design goals and assert the programming objectives (e.g., thread based parallelism) provisionally without committing fully and early-on to constructing the PL structures and details. Why? Because those PL structures and details are likely to change and evolve as the target program is refined toward a final implementation. That potential change and evolution is difficult in the PL domain because of the subtle interdependencies among the PL structures and details. For example, data flow dependencies in the context of scopes and routines make code movement and revision quite complex. The ideal solution would allow each atomic design objective or intended design feature to be introduced one at a time. These design objectives and intended design features would imply and constrain the eventual code but not immediately construct it. It is far easier to back out of or alter design objectives and intended design features than it is to alter the eventual code forms that express those design objectives and intended features. In the ideal solution, previously asserted provisional commitments could be

altered before they are cast into concrete code. And this idea is the essence of DSLGen™.

DSLGen™ allows the construction of a logical architecture (LA) that levies minimal constraints on the evolving program and explicitly defers generating programming language structures (either concrete or abstract) early on. That is, initially the LA will constrain only the decomposition of a computation into its major (and natural) organizational divisions (which are called *natural partitions*) omitting any PL details of the programming routine structure or PL details of those major organizational divisions. There is no information on control structure, routines, functions, threads, parametric connections, data flow connections, machine units, instruction styles, parallel synchronization structures and so forth. All of that is deferred and added in step by step as the generation process proceeds. In fact, the LA will be revised and evolved step by step via the encapsulation of individual design features, each of which will further constrain the final expression of the target program.

A. *Associative Programming Constraints and the LA*

DSLGen™ builds the LA out of a new kind of representation element – an *Associative Programming Constraint* (*APC*). APCs are partial and provisional constraints on the target computation. They do not fully determine the target implementation. The motivation for APC's is analogous to the motivation for modifiers in natural language. That is, an APC is a modifier of a domain specific expression (e.g., a convolution expression) and it implies some distinct (possibly global) design feature in the eventual programming language (PL) implementation form of that domain specific expression. For example, the APC might provide the "nominal" form of the loop or loops required to perform the computation. However, this is only a partial specification of the PL implementation form because it does not determine the context or even the concrete implementation form of the loops. There are many open questions unanswered by a singular APC. For example: Are the implementation loop or loops partitioned into pieces? And are those loop pieces organized into thread routines or re-expressed as Intel's SSE vector instructions (e.g., PMADD instructions)? If SSE instructions, what triggers the reorganization necessary to reform the weight values into vectors? And so forth. Thus, a singular APC is unlikely to be sufficient to generate the desired implementation for the target implementation. It is unlikely that code written directly from a singular APC will be the same as the eventual code of the generated target implementation. Too many other design features (represented by other APCs) will be needed to fulfill the requirements imposed by the user's description of the execution platform features to be exploited in the target routine. It is much more likely that a number of APCs will be required to fully specify the PL implementation. Furthermore, there is seldom a one to one mapping between an APC and a programming language abstraction in the target implementation. More generally, the mapping is many to many. Moreover, like modification structures in natural language, these APCs will need to be formulated into a structure (i.e., the logical architecture) that captures the interrelationships among them. For example, a partitioning APC may modify a loop APC and therefore imply addition features of the PL loop implementation.

APCs come in two major varieties: *Iteration constraints* and *partition constraints*. For example, a *loop constraint* (a subclass of iteration constraint) might specify "i" and "j" to be provisional indexes of a matrix "a". They might have ranges of [0,(m-1)] and [0,(n-1)], respectively. And related to this loop constraint, for example, might be a partition constraint (e.g., Edge1) that modifies the loop and specifies the subdivision of that loop in which (i==0). In other words, the Edge1 partitioning constraint implies that the loop over i is degenerate and will refine to the operation that is just the body of the loop. Nothing further about the implementation is determined by these constraints.

Conventionally, one tends to think of "constraints" as being represented by some kind of formulaic expression (e.g., a predicate calculus expression). However, while formulaic expressions do play a role in some APCs (e.g., partition APCs will have a so-called "partitioning condition" expression), APCs also have several additional representational facets and features. Operationally, they are CommonLisp Object System (CLOS) objects that are associated with elements of the INS and initially arise via translation of the INS. They imply something about the eventual PL implementation of that INS expression by their existence and interrelationships. But beyond that, because they are problem domain entities and not programming language abstractions, they also may have problem domain knowledge features or properties that are useful to the generator. For example, image "edges" have the domain property of being "lightweight" computations and that property may be employed by the generator to decide upon the details of thread designs. Later in this paper, we will see that the edge loops in a thread design are batched into a single thread rather than each having their own thread. The "lightweight" domain property is used heuristically by the generator to make that design decision. Similarly, image center partitions are known to be "heavyweight" computations because there are often many individual computations to be performed and because the individual computations are often fairly complex. That domain property will be used by the generator to decide to slice the center partition into computational slices and to assign each to its own parallel thread.

The constraining affects of APCs will likely need to be altered and refined as the generation process proceeds and this will be effected by altering and specializing the definitions of the constraints. For example, an edge partition that modifies (i.e., is associated with) a loop, is likely to cause component definitions specific to that edge to be specialized thereby altering the loop's beginning index value, ending value and increment. Those specializations may lead to loops completely evaporating, which will happen in the forthcoming example. Furthermore, some specialized definitions (e.g., the definition of the weight coefficient for a neighborhood specialized to an edge) may cause simplification of a loop's body, which might compute the

value of a single edge pixel. In the forthcoming example, the code to compute an edge pixel will simplify to 0. Such simplification is effected via DSLGen™'s built-in partial evaluation system (see also [21]).

B. Creating and Evolving the Logical Architecture

Loop APCs are created and propagated over the INS structure (somewhat analogously to APL's method of loop introduction and placement). In the course of that process, the creation of a loop APC may trigger the creation of partition APCs by a process we will describe in a moment. APCs are combined in several ways. For example, the operational effect of combining equivalent APC sets, is to merge equivalent iterations (e.g., two loop APCs) or to adapt two slightly different computational cases to a single interation scheme (e.g., the computations introduced by two separate domain specific operators such as convolution operators). APCs can be split in two, reorganized into groups that imply future design features and revised to incorporate one or more elective design features (e.g., multicore, threaded design). Furthermore, partition sets of logically orthogonal APCs can be combined by a cross product operation to produce a new set of partitions. Not until later in the generation process are the APCs actually applied by replicating and cloning the INS into multiple distinct forms specialized for different partitions of the implementation. These specialized INS clones are the precursors to the actual PL expression of the target implementation.

Specialized versions of APCs may be created by subclassing, thereby allowing other kinds of architectural factorings. For example, an image center partition may be specialized to a slice of an image center in anticipation of computing slices of the image center in parallel threads.

To provide a concrete context in which to discuss the LA and its representational elements, we will introduce a problem domain, a domain specific language for that problem domain and a concrete example.

IV. THE PROBLEM DOMAIN AND AN EXAMPLE PROBLEM

The initial problem domain treated by DSLGen™ is digital signal processing (DSP) and includes problems that range from signal and image processing to neural networks to pattern recognition plus a rich set of related problems. The domain specific language used to express the INS is based on the Image Algebra (IA) [28].

As an example computation, we develop a program that performs Sobel edge detection on a grayscale image (i.e., where the pixels are shades of gray). Such a program would take, for example, the image "a" in Fig. 1 as input and produce the image "b" in Fig.2 as output. The output image has been processed so as to enhance (line) edges of items in the image by the Sobel edge detection method.

Each black and white pixel b[i,j] in the output image "b" is computed from an expression involving the sum of products of pixels in a neighborhood (e.g., sp, of type *iatemplate)* surrounding the a[i,j] pixel and the coefficients defined by that neighborhood (e.g., sp). This is called a *convolution* of a matrix with a template (or neighborhood). In the IA, a convolution is designated by the ⊕ operator, e.g., (a ⊕ sp). In the following examples, s and sp will designate instances of the class iatemplate. Mathematically, the Sobel computation is defined as

$$\{\text{Forall}_{i,j}\,(b_{i,j} : b_{i,j} = \text{sqrt}((\textstyle\sum_{p,q}(w(s)_{p,q} * a_{i+p,j+q})^2 + \sum_{p,q}(w(sp)_{p,q} * a_{i+p,j+q})^2)\} \quad (1)$$

where i and j are indexes that range over the matrices a and b; p and q are indexes that range over the iatemplate neighborhoods s and sp; and the coefficients of the neighborhood (which are also called *weights*) are defined by the function "w". For Sobel edge detection, the weights are all defined to be 0 if the center pixel of the neighborhood corresponds to an edge pixel in the image (i.e., w(s) = 0 and w(sp) = 0), and if not an edge pixel, they are defined by the s and sp neighborhood weights shown in (2). It is convenient to index the neighborhoods in the DSL from -1 to +1 for both dimensions so that the current pixel being processed is at (0, 0) of the neighborhood.

$$w(s) = \begin{matrix} \\ P \\ \\ \end{matrix}\overset{\overset{Q}{\overbrace{}}}{\begin{matrix} -1 & 0 & 1 \end{matrix}}\begin{matrix} -1 \\ 0 \\ 1 \end{matrix}\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \qquad w(sp) = \begin{matrix} \\ P \\ \\ \end{matrix}\overset{\overset{Q}{\overbrace{}}}{\begin{matrix} -1 & 0 & 1 \end{matrix}}\begin{matrix} -1 \\ 0 \\ 1 \end{matrix}\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad (2)$$

Since an implementation of this computation for a parallel computer may not be organized like the mathematical formula, it is useful to represent this specification more abstractly because such abstractions can defer the implementation and organization decisions and thereby allow the computation (i.e., what is to be computed) to be specified completely separately and somewhat independently from the implementation form (i.e., how it is to be computed). Thus, the abstract computation specification is independent of the architecture of the machine that will eventually be chosen to run the code. Choosing a different machine architecture for the implementation form without making any changes to the specification of the computation (i.e., the what), will automatically generate a different implementation form that is tailored to the new machine's architecture. More to the point, porting from one kind of machine architecture (e.g., machines with instruction level parallelism like Intel's SSE instructions) to a different kind of machine architecture (e.g., machines with large grain parallelism such as multi-core CPUs) can be done automatically by only making trivial changes to the machine specifications and no changes to the computation specification (i.e., the what). The publication form in [28] for the Sobel Edge detection mathematical formula (1) is based on the Image Algebra domain specific language (DSL). Re-expressing the formula (1) in the Image Algebra gives a first cut at the INS for the Sobel example:

$$b = [(a \oplus s)^2 + (a \oplus sp)^2]^{1/2} \quad (3a)$$

Of course, the INS will need some declarations for a, b, s, sp, etc.:

```
(DSDeclare IATemplate s :form (array (-1 1) (-1 1))
        :of DSNumber)
(DSDeclare IATemplate sp :form (array (-1 1) (-1 1))
        :of DSNumber)
(DSDeclare DSNumber m :facts ((> m 1)))
(DSDeclare DSNumber n :facts ((> n 1)))
(DSDeclare BWImage a :form (array m n) :of BWPixel)
(DSDeclare BWImage b :form (array m n) :of BWPixel)
```

$$b = [(a \oplus s)^2 + (a \oplus sp)^2]^{1/2} \quad (3b)$$

m and n are assumed to be user defined. The DSL type declarations (e.g., IATemplate, BWImage, etc.) define CLOS types that will eventually refine to C types. The ":facts" keyword denotes a conjunction (i.e., list) of facts pertinent to the declared item (e.g., m) and will be used to infer, for example, that"(i==(m-1))" is false when "(i==0)" is true. Beyond (3b), we will also need some definitions for w of s and sp equivalent to (2) as well as for $\oplus$. These will be defined later.

This DSL is the basis of the Implementation Neutral Specification (INS) in the examples used throughout the remainder of this document. A full description of the IA used by DSLGen™ is beyond the scope of this paper (see [28]) but a few comments are in order. The IA is much like APL in the sense that IA specifications eschew the use of explicit looping constructs allowing loops to be implied by IA operators and data structures. The generator will introduce implied loops as constraints and, through the manipulation, combination and propagation of these constraints, will determine the relationships between IA expressions and loops. The initial form of the LA arises during this process.

Figure 1. Input Image a

In DSLGen™, the Image Algebra is adapted to a more utilitarian, LISP based syntax with prefix operators, without the pretty symbols (e.g., the convolution operator $\oplus$ becomes a Lisp symbol), and with the w functions in (1) becoming so-called *Method-Transforms* (*MT*), which rewrite *Abstract Syntax Tree* (*AST*) subtrees. MTs look superficially a bit like object oriented methods with a pattern (i.e., the MT's *left hand side* or *lhs*) as the analog of a method's parameter sequence and a pure functional expression *right hand side* (*rhs*) as the analog of a method's body. MTs will be an important component of the *intermediate language* (*IL*) by which provisional but malleable low level definitions are expressed. For example, w of the neighborhood sp is an MT expressed as:

```
(Defcomponent w (sp  #. ArrayReference ?p ?q)
  (if (or (== ?i  ?ilow) (== ?j  ?jlow)
          (== ?i ?ihigh) (== ?j ?jhigh)
            (tags (constraints partitionmatrixtest edge)))
      (then 0)
      (else (if (and (!= ?p 0) (!= ?q 0))
                (then ?q)
                (else (if (and (== ?p 0) (!= ?q 0))
                          (then (* 2 ?q))
                          (else 0)))))))                    (4)
```

where ArrayReference is the name of a shared pattern that together with the generator variables ?p and ?q will gather elements of the generator's context of the convolution operation involving the neighborhood sp. Part of this context will come from a pixel array reference in an AST (e.g., a[i,j]) that will result in the binding of the loop index variables (e.g., i and j) to the pattern variables ?i and ?j, the image matrix name a to ?a and the expressions defining the upper and lower ranges of those loop indexes to ?ihigh, ?ilow, etc. The remainder of the lhs pattern after ArrayReference will bind ?p and ?q to the loop index names used by the inner convolution loops over the neighborhood designated by sp. This generator context is the mechanism by which the generator preserves the connection between elements of the problem domain specification (e.g., a convolution expression) and the constituent elements of the evolving programming language domain implementation (e.g., the details of the image and neighborhood loops that will eventually implement that problem domain expression). Thus, the problem domain knowledge is used as a meta-knowledge context to relate high level goals (e.g., build a convolution computation) to the low level programming elements (e.g., loop building blocks for the image and neighborhood loops) out of which the implementation code will be built to achieve that high level goal.

The "tags" expression designates a property list for the OR conditional expression, which in (4) provides the user supplied domain knowledge that the OR expression is a *partitioning condition* for this computation that will identify *edge* partitions and by implication, a non-edge (i.e., *center*) partition. Problem domain concepts like "edge" and "center" play a key role in the logical architecture for the target computation and beyond that, in imposing design pattern frameworks onto a logical architecture. Heuristic rules based on domain concepts are the mechanisms whereby DSLGen™ chooses a design pattern framework to introduce PL structures and clichés (e.g., coordinated routines, synchronization patterns and thread management clichés) and maps the LA into the structures and clichés of that design pattern framework.

The opportunity for such domain specific heuristic rules is open ended, especially given the rich variety of possible semantic subclasses of partitions. Different problem examples may introduce other domain semantics. For example, in the matrix domain, the semantic subclasses include *corners* (e.g., corners are special cases in partitioning image averaging computations); *non-corner edges* also used in image averaging; *upper and lower triangular* matrices, which are used in various matrix algorithms; *diagonal* matrices; and so forth. By contrast, in the data structure domain, domain subclasses include *trees*, *left and right subtrees*, *red and black nodes*, etc. In general, problem domain concepts drive program generation.

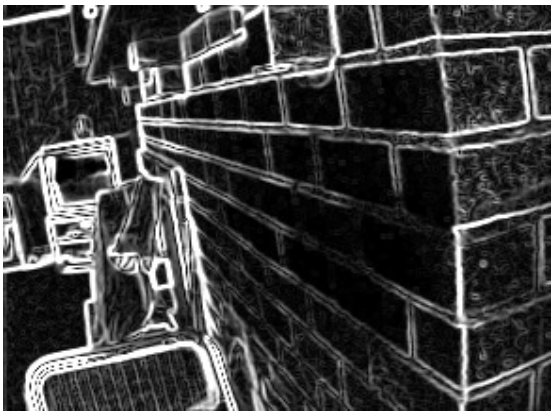

Figure 2. Output Image b

V. The Design Representation System And Its Operation

The first iteration of the logical architecture for the Sobel example is shown conceptually in Figure 3. Loop constraints are CLOS objects that keep track of loop indexes, loop nesting and the logical description of the loop, which comprises logical assertions and precursors thereof that constrain or restrict the loop in some way. For example, the MT definition of Partestx of s is IL manufactured during initial INS translation process that transforms an INS expression on images into a partially translated INS expression on pixels. Partestx of s is a precursor logical assertion that will eventually refine to a concrete partitioning condition for some (not yet decided upon) partition. Partestx of s will eventually be refined to a concrete expression such as "(i==0)" in the context of a particular partition-based computation (e.g., Edge1). And the addition of "(i==0)" to the loop constraint will change the form of the C code that is eventually generated for that partition by causing the loop over "i" to evaporate and possibly allowing the body of the loop to be simplified. In the chosen example, the bodies of edge loops undergo significant simplification.

Operationally, Partestx is a closure over one of the disjuncts (e.g., (== ?i ?ilow)) in the OR expression in (4) and the translation context bindings (e.g., ((?i i) (?ilow 0)) at the time of Partestx formation. That translation time will be when an expression like "(a ⊕ s)" is being translated and a provisional loop constraint is being introduced and propagated to the "⊕" level expression.

As loop constraints are introduced, propagated and combined (e.g., providing loop sharing for separate computations), DSLGen™ provides machinery for recording design decisions (e.g., discarding unneeded loop indexes) via dynamically generated transformations that will be applied periodically to synchronize the overall design. That is to say, several provisional loop constraints with provisional index names will be introduced as the generator walks over the expression tree. Only later, as these individual loop constraints propagate up the tree, does the generator discover that they can be combined thereby allowing one loop to replace two separate loops and thus, optimizing the computation. However, residual occurrences of no longer valid loop index names still exist in the contexts where they were introduced. To allow this to be fixed up, the generator dynamically creates transformations that incorporate the design decisions (e.g., discard loop index name i1 and replace it with i2). Later, the generator walks over the expression tree applying these fix up transformations and thereby synchronizes the overall expression.

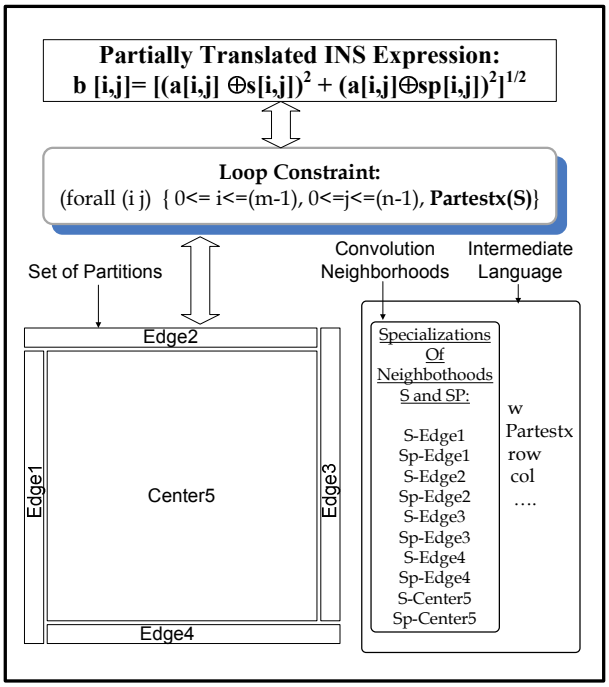

Figure 3. Initial logical architecture of example

The loop constraint in Fig. 3 is associated (via the "fat" double headed arrow) with a partially translated INS expression. Operationally, this association is effected by the loop constraint appearing on the INS's tags list as a property of the INS. Generally speaking, the loop constraint may be associated with a set of partitioning constraints such as the Edge1, Edge2, Edge3, Edge4 and Center5 (i.e., the CLOS objects) of this example. They indicate a partial and provisional decomposition of the loop, where each

decomposition body eventually will be formed from a cloned and specialized version of the associated INS expression. But DSLGen™ does not perform the decomposition yet, because as the implementation design evolves, the partitioning is almost certain to change before it is cast into code. The partitioning implied by the set of partition objects is sort of a "to do" list and a "to do" list that will likely change before it is turned into code. However, this future cloning and specialization will be accomplished by using a set of newly formed specializations of s, sp and their IL. For example, the specialization of a specific neighborhood (e.g., sp) and its IL (e.g., w) for a specific partition constraint (e.g., edge1) is formed by assuming a truth value for the partitioning condition of the partition constraint and partially evaluating the IL definitions under that assumption. For example, for Edge1, the MT definition of w of sp, in (4), would partially evaluate to the new MT definition, w of sp-edge1 show in expression (5):

(Defcomponent w (sp-edge1 #.ArrayReference ?p ?q) 0) (5)

The LA is malleable so that DSLGen™ can incrementally introduce design features by a process called *Design Feature Encapsulation* (*DFE*). DFE will revise IL definitions, extend and reorganize partition sets and occasionally even revise some of the DSLGen™'s own transformations that define the overall generation and programming process (e.g., when introducing instruction level parallelism).

To ground the LA concept a bit more in concrete reality, Fig. 3a shows what a domain engineer who is extending or debugging a domain model would see by using the Architecture Browser (AB) tool of DSLGen™. It shows a concrete example of an LA that is roughly analogous to the conceptual version of Fig. 3. The correspondences with the examples chosen for this article are not exact for reasons that are not relevant to this paper. However, the neighborhood names "SX" and "SPX" in Fig. 3a are obviously the analogs of the names of "s" and "sp" used in the earlier examples of this paper.

In Fig. 3a, the left hand panel of the AB shows the architectural structure associated with (i.e., modifying) the loop constraint Loop2d5, which is of type "loop2d", where Loop2d5 is the fifth loop2d instance that DSLGen™ has generated so far. The Loop2d5 constraint is modified by a partition set (partitionset3) that contains five partition APCs (i.e., edge11, edge12, edge13, edge14 and center15). For the edge11 partition, whose substructures have been opened for examination, there are two domain variables (i.e., the sx-0-edge11 and spx-0-edge11 neighborhoods) that are specialized to the edge11 partition. The sx-0-edge11 neighborhood variable has been opened to reveal the component definitions (i.e., method transforms) that have been specialized to it. These method transforms will eventually refine to concrete code within the target program context that is dealing with edge11.

sx-0-edge11 and spx-0-edge11 correspond to our conceptual example's specialized neighborhoods s-edge1 and sp-edge1. The sx-0-edge11 and spx-0-edge11 neighborhoods have been combined because they are both

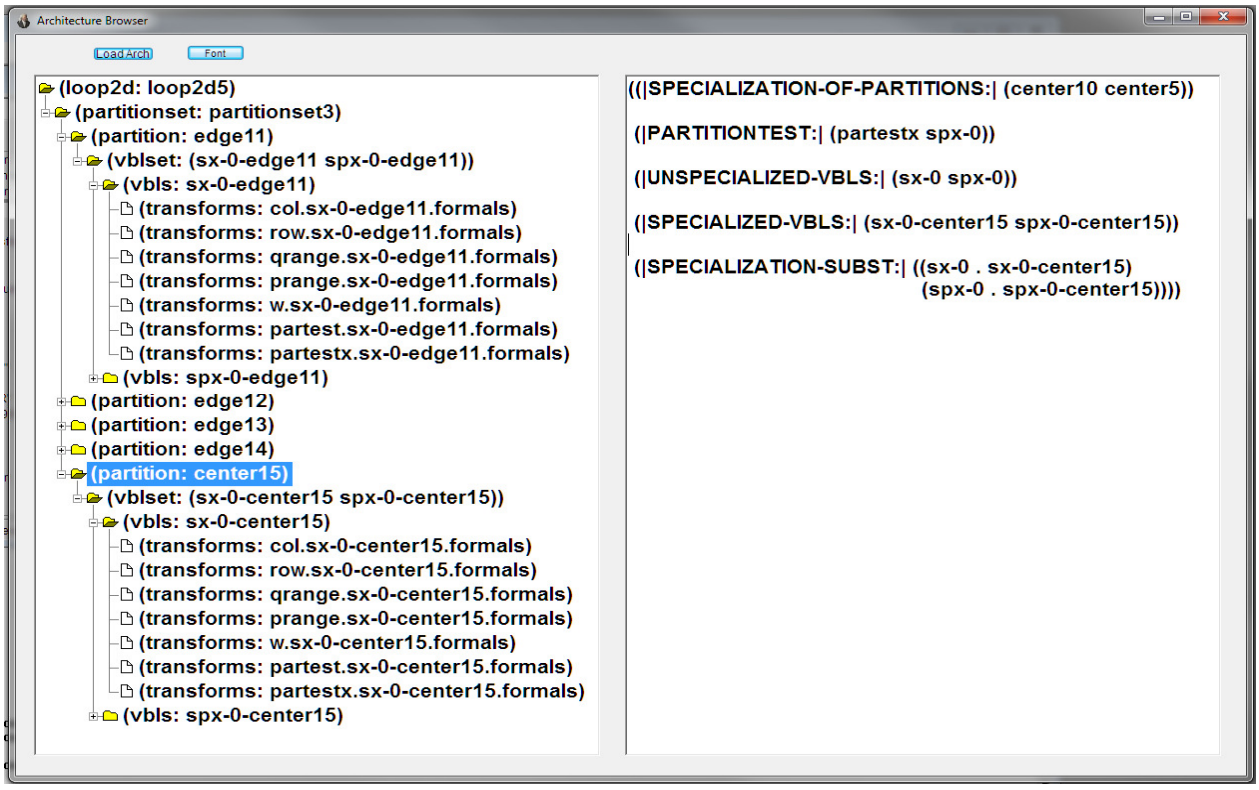

Figure 3a. Browsing the LA

specialized on the same partitioning condition. That is, the generic IL forms (Partestx sx-0-edge11) and (Partestx spx-0-edge11) will both refine to the same concrete form (e.g., "(== idx1 0)").

The method transforms organized within these partition structures (e.g., "w.sx-0-edge11.formals") represent the specialized components that will be used to generate code when they are eventually inlined (i.e., in a phase named "formals"). These components are specialize to their specific partitions via the process describe above and illustrated for an edge partition by formula (5). The triple dotted transformation names identify the CLOS object that defines the transformation. The triple dotted name parts comprise the method or transformation name (e.g., "w"), the CLOS object where the transformation object is stored (e.g., sx-0-edge11) and the generator phase name for which the transform is enabled (e.g., "formals" for MT's). All transformations are enabled (i.e., will be tried and can possibly fire) only during their named phase. Thus, the overall DSLGen™ process is defined by a list of phase names (which are user definable and extensible) and a package of transformations for each phase. Phases define high level generation tasks like build scope structure, process declarations, do initial type inference, create logical architecture, build synthetic architecture, etc.

The right hand panel of Fig. 3a shows the key fields of the **selected partition** constraint (i.e., the "center15" partition). Interestingly enough, it reveals that the center15 partition was created by merging the center5 and center10 partitions, which arose from the two different convolution expressions of formula (3a). Since the partitioning condition of both will refine to the same concrete code (e.g., "(== idx1 0)"), they can be combined. The operational result of that combination is that the two convolution computations can share a single loop thereby avoiding two passes over the image.

The initial logical architecture illustrated in Figs. 3 and 3a captures only those design features and structures that are inherent to the computation specified by the user. What remains to be determined are the elective design features that arise because of the user's specification of the implementation architecture features that he or she wants to exploit in the final implementation. The next section will look at how the elective design features are created and incorporated in the LA by a process of Design Feature Encapsulation.

A. *Design Feature Encapsulation*

For our example, let us use an EXPS of "((PL C) Mcore (Threads MS) (LoadLevel (SliceSize 5)))." This specifies: 1) C is the output language, 2) the target is a multicore machine that exploits threaded parallelism using Microsoft's thread library and 3) the design should decompose the computation by slicing up some unspecified heavyweight computation using 5 unspecified units per slice. In the example, the LA specifics will be used to disambiguate what is being sliced up (e.g., Center5) and what kind of units comprise a slice (e.g., matrix rows).

In figure 3, we have already seen a simple example of DFE where IL definitions are specialized to specific logical partitions of a target computation. These specializations will cause computations along the matrix edges to simplify to a single loop that assigns 0 to pixels of that edge. Another simple example of DFE is mapping from IA neighborhood style indexing to C style indexing. IA style indexing ranges from –n to +n for a (2n+1) by (2n+1) neighborhood so that the center pixel is at (0,0). In contrast, the C language (i.e., the chosen output language) arrays range from 0 to 2n. The indexing DFE is accomplished by algebraic manipulation of the right hand side (i.e., the MT body) of IL involving neighborhood loop indexes, which relocates instances of those loop indexes appropriately.

However, one of the most powerful examples of DFE is the introduction of elective architectural design features that alter the form and relationships within the implementation across a broad set of coordinated routines, data structures and possibly even parallel processes. This is accomplished by the use of *synthetic partitions*, which extend the notion of natural partitions by adding elective design feature constraints that are implied by the EXPS.

1) *Introducing Synthetic Partitions*

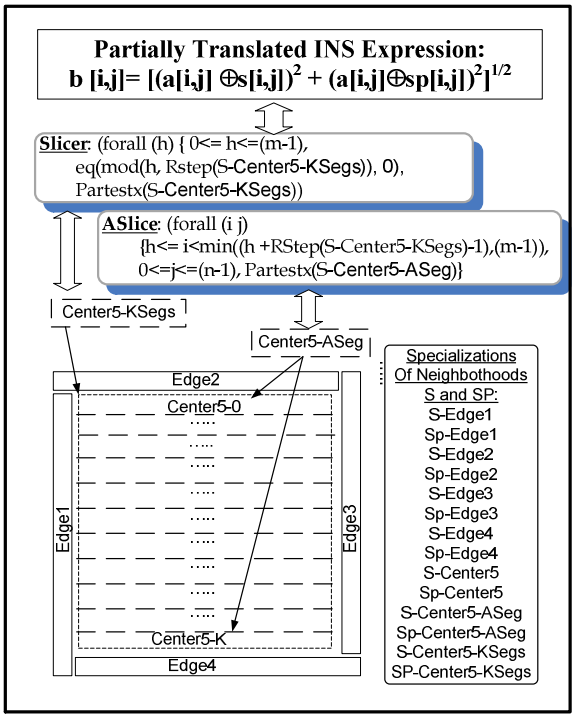

Figure 4. Revised logical architecture of example

In DSLGen™, the generation process is divided into named phases, each of which has a narrowly defined generation purpose. The phase most relevant to the introduction of wide ranging elective design features is the Synthetic Design phase. During the Synthetic Design phase, the generator introduces design features (e.g., via synthetic partitions as

well as synthetic loop APCs) that will constrain the evolving LA to be much more specific to a design for some specific execution platform. These synthetic partitions imply implementation structures that exploit high capability features of the execution platform and that, when finally re-expressed in a form closer to code, may have a global and coordinated affect across much of the LA (e.g., via multiple routines that coordinate the use of multicore parallel computation). The Synthetic Design phase operates on the logical architecture to revise and reorganize loop APCs, to reorganize the partitions and probably (depending on the execution platform spec) to create synthetic partitions that are consistent with one or more design frameworks. These design frameworks introduce the implementation level details (e.g., thread and synchronization management as well as low level programming clichés) to be integrated into the evolving target program. How does the Synthetic Design phase affect the LA of Fig. 3?

Our chosen example EXPS requires that the computation should be load leveled (i.e., sliced into smaller computational pieces) in anticipation of formulating the computation to run in parallel threads on a multicore platform. Given the EXPS requirements, Fig. 4 shows the revised logical architecture arising from the example EXPS. The synthetic partitions are denoted by dashed boxes. The load leveling requirement will engender two synthetic partitions (e.g., Center5-KSegs and Center5-ASeg) that respectively express the design feature that assumes the center partition (i.e., Center5) is decomposed into smaller pieces and the design feature that implies code that will process each of those smaller pieces.

Simultaneously, in Fig. 4, the loop constraint from Fig. 3, is reformulated into two loop constraints (i.e., Slicer and ASlice) that will be required by the synthetic partitions Center5-KSegs and Center5-ASeg. This synthesis process also introduces versions of the neighborhoods S-Center5 and SP-Center5 specialized for Center5-Ksegs and Center5-Aseg and generates specialized the IL for each. The step size of the Slicer loop is inferred from information in the EXPS or from a default if the EXPS is silent on the subject. The step size is represented by the IL expression "Rstep(S-Center5-Ksegs)" in Fig. 4. For the example, we have chosen a step size of 5. Using this step size (with an inferred dimension of "rows"), the code engendered by Slicer will dynamically compute a new range for each instance of the ASlice loop. Thus, the Aslice loop in the first thread will have a range of [0,min(4,(m-1))], the second [5, min(9,(m-1))] , the third [10, min(14,(m-1))] and so forth.

2) Cloning and Specialization

At this point, DSLGen™ is ready to create explicit instances of the separate computation cases so that those cases can be moved to the correct places in the emerging global design architecture. This is accomplished by creating *clones* of the APCs that are specialized to the various partitions (i.e., cases). Fig. 5 illustrates the specialized clones that will be created from the synthetic logical architecture of Fig.4. These clones will supply the design features specific to the essence of the computation (e.g., specs for loops and for computational steps). They will be combined with a *design framework* that will supply the overall architecture and design features specific to the elective requirements of the computation implementation (e.g., patterns of cooperative routines, pattern of synchronization and even low level program code supporting both). More specifically, the clones will be used to fill in holes (i.e., undetermined parts) of the PL based design framework.

A design framework is roughly a formalization of the "gang of four's" notion of a design pattern [19]. It is basically a large scale skeletal code pattern (e.g., a pattern of coordinated routines) with holes that expect certain kinds of LA elements. For example, some holes will expect loop APCs associated with some version of the INS that is specialized for lightweight computations (e.g., image edge loops shown in Fig. 5 as callout 5-05 and their respective INS clones shown as callout 5-03). Others will expect a loop APC with a version of the INS specialized to heavyweight computations (e.g., a slice of an image center, which is shown as callout 5-10) and its corresponding loop APC (e.g., Aslice shown as callout 5-02). Yet others (for this specific example) may expect synthetic APCs that are tailored to elective design features such as a loop that slices up the data structure associated with the heavyweight computation (e.g., a loop like Slicer in Figs. 4 and 5, where Slicer is shown as callout 5-01). The process of combining the clones of Fig. 5 with a design framework is described in some detail in the next section.

3) Merging Design Patterns with a Logical Architecture

At this point, DSLGen™ is ready to add in the PL level details (e.g., a pattern of interrelated routines, parametric plumbing, thread management clichés and protocols of specific thread libraries) by mapping the LA into a PA through use of a design pattern framework. DSLGen™ allows for a library of design pattern based frameworks (i.e., objects with associated PL-like skeletons), each of which represents some reasonably small combination of related elective design features. Additionally, each such framework has a set of holes containing *protocol expressions* (indicated by **embolden designators**) that specify elements of the LA that should be substituted for the protocol expressions. The specific design framework chosen by DSLGen™ is determined by the LA's combination of architectural features as well as problem domain properties of the target computation. For example, the convolution operations in the problem domain specification of our example (i.e., expression (3a)) reveal that each output pixel computation is independent of other output pixel computations, which is a requirement of the chosen design framework. Such a property can be determined by examining only the types and structure of the domain specific expressions. If there had been some interdependencies among separate pixel computations, it would have changed both the course of the synthetic design process and the possible design framework(s) that could apply. In such a case, the whole structure of the design framework would have been different and in particular, the patterns of synchronization would have been different.

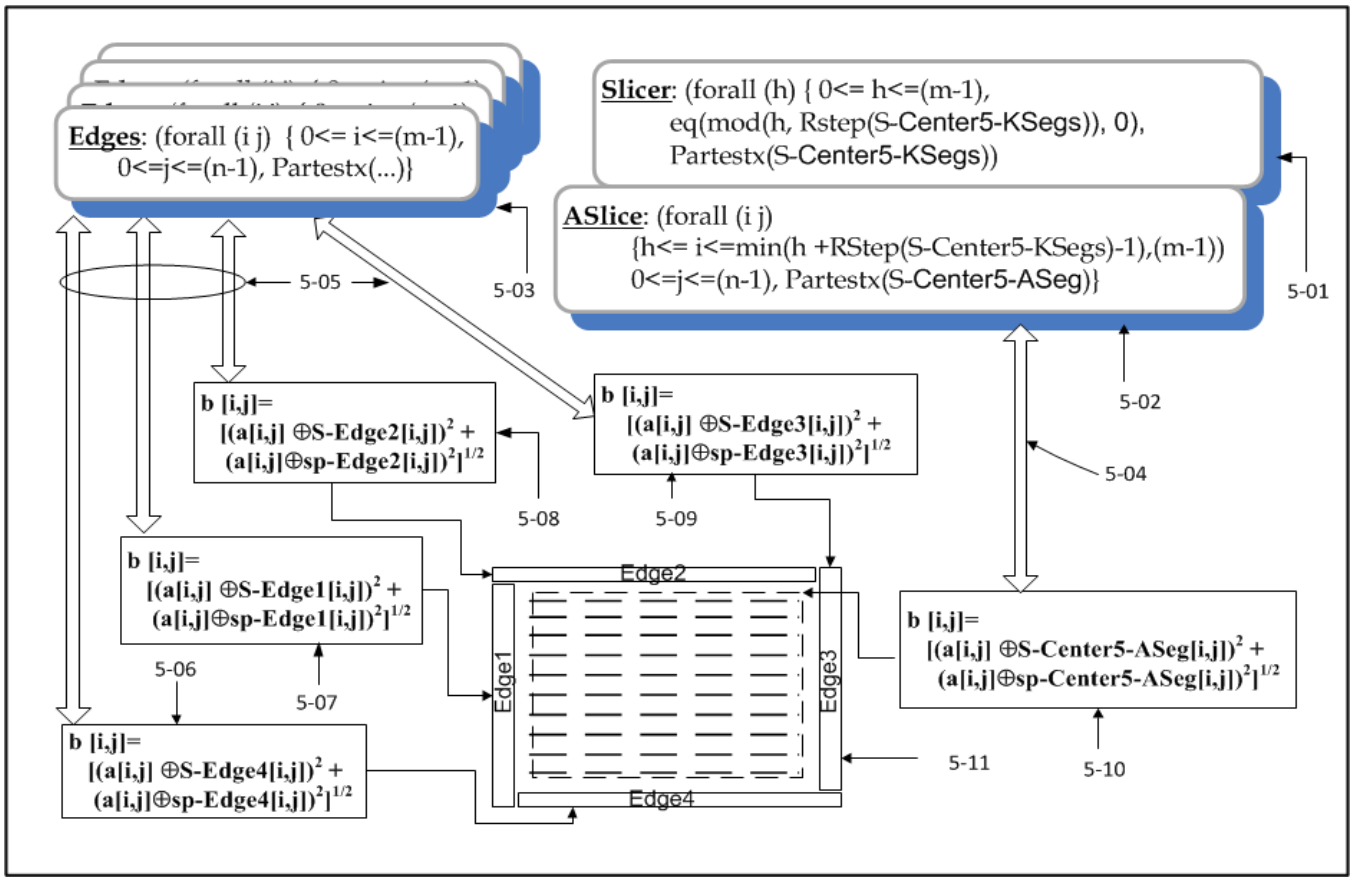

Figure 5. Specialized Clones

The holes in the design framework are designed to receive computational payloads from the LA (e.g., partition specific computations). For example, a particular framework might be designed to receive partitions such as image edges that are "probably" order n computations (i.e., lightweight computations) as well as to receive partitions such as image centers that are "probably" order n squared computations (i.e., heavyweight computations). Such a framework might introduce a set of cooperating PL routines and the parametric plumbing among those routines, where the plumbing may include some "holes" that will receive data items specific to the INS. There may be additional PL design features included, such as synchronization patterns for parallel computation and detailed thread control clichés. But the framework is agnostic about its payload. It says nothing about exactly what kind of a computation is occurring in its holes. That computational payload information will be supplied by the logical architecture.

So, based on the example LA plus specific features required by the EXPS, DSLGen™ will search its design pattern data base for a design pattern meeting these criteria. It finds one with the following skeletal PL framework:

```
void  ?managethreads ( )
  { HANDLE threadPtrs[200];
    HANDLE handle;
    /* Launch the thread for lightweight processes. */
    handle = (HANDLE)_beginthread(
                 &?DoOrderNCases , 0, (void*) 0);
    DuplicateHandle(GetCurrentProcess(), handle,
          GetCurrentProcess(),&threadPtrs[0],
          0, FALSE, DUPLICATE_SAME_ACCESS);
   /* Launch the threads for the slices of heavyweight
      processes. */
    {handle = (HANDLE)_beginthread(& ?DoASlice , 0,
                 (int) (Idex ?SlicerConstraint)  ) ;
    DuplicateHandle(GetCurrentProcess(), handle,
          GetCurrentProcess(),&threadPtrs[tc],
          0, FALSE, DUPLICATE_SAME_ACCESS);
     tc++; } (tags (constaints ?SlicerConstraint))
     long result = WaitForMultipleObjects(tc, threadPtrs,
                          true, INFINITE); }                (6)

void ?DoASlice (int  (Idex ?SlicerConstraint))
      {{ ?ins } (tags (constraints ?ASliceConstraint))
      _endthread( );  }                                     (7)

 void ?DoOrderNCases  ( )
      {?OrderNCases
       _endthread( ); }                                     (8)
```

Associated with the class of this design pattern is a CLOS method whose job is to find key elements in the LA and bind them to pattern variables (e.g., **?ins** and **?SlicerConstraint**); invent and bind unique names for routines (e.g., "SobelCenter8" might be invented for **?managethreads**); clone and specialize the INS to specific partitions (e.g., by substituting sp-Edge1 for sp); and instantiate the skeletons with the bindings. Notice that the design skeletons are agnostic as to what their computational payload is going to be. Further, there are no PL like connections (e.g., calls to PL routines) between the design pattern skeletons and anything in the LA. The only requirements of the design pattern are that the LA has partitions that represent lightweight processes that can be batched in a single thread (e.g., edges) and a heavyweight process (e.g., a center) that is partitioned into a slicer partition and an implied set of slicee partitions. These requirements are determined by **domain logic**, that is, logical rules operating on problem domain information (e.g., properties of edges) rather than PL information.

Space limitations preclude showing the full step by step expansion of all these skeletal routines but the thread routine that batches the edge partitions (**?DoOrderNCases**) is reasonably short and is interesting in that the edge loops will drastically simplify when in-lined and partially evaluated. Instantiating with cloning and specialization produces:

```
void SobelEdges9( )
 { /* Edge1 partitioning condition is  (i=0) */
  {for (int j=0; j<=(n-1);++j)
     b [0,j]= [(a[0,j] ⊕ s-edge1[0,j]) 2 +
               (a[0,j] ⊕ sp-edge1[0,j]) 2] 1/2}
   /* Edge2 partitioning condition is  (j=0) */
   {for (int i=0; i<=(m-1);++i)
     b [i,0]= [(a[i,0] ⊕ s-edge2[i,0]) 2 +
               (a[i,0] ⊕ sp-edge2[i,0]) 2] 1/2}
   /* Edge3 partitioning condition is  (i=(m-1)) */
   {for (int j=0; j<=(n-1);++j)
     b [(m-1),j]= [(a[(m-1),j] ⊕ s-edge3[(m-1),j]) 2 +
                   (a[(m-1),j] ⊕ sp-edge3[(m-1),j]) 2] 1/2}
    /* Edge4 partitioning condition is  (i=(n-1)) */
   {for (int i=0; i<=(m-1);++i)
     b [i, (n-1)]= [(a[i, (n-1)] ⊕ s-edge4[i, (n-1)]) 2 +
                    (a[i, (n-1)] ⊕ sp-edge4[i, (n-1)]) 2] 1/2}
    _endthread( ); }                                          (9)
```

Notice that in expression (9), partial evaluation plus inference has caused one of each pair of the edge loops in (9) to evaporate and the edge index max or min values (e.g., 0 or (m-1)) to appear in one of the index positions in the array expressions. In the implementation, these loop refinements occur concurrently with the inlining of the IL definitions (see the following section) but in the name of simplicity, showing it here shortens (9) and makes it easier for the reader to understand.

While the expansion of the ?DoASlice routine is longer than SobelEdges9, it is important because is shows the default partition specialization (i.e., the center slice partition). It is populated with the Aslice loop constraint plus the INS specialized to S-Center5-ASeg and SP-Center5-ASeg. The example code is shown with a slice size of 5 but alternatively, it could be declared by the user to be a parameter. Before inlining the IL definitions, the ?DoASlice routine is specialized to:

```
void SobelCenterSlice10 (int h)
 {
  for (int i=h; i< =min((h+ 4),(m-1)); ++i)
      {for (int j=1; j<=(n-2); ++j)
          {b [i,j]= [(a[i,j] ⊕ s-center5-Aseg[i,j])^2 +
                     (a[i,j] ⊕ sp-center5-ASeg[i,j])^2]^1/2 } }
           _endthread( );   }                                  (10)
```

Like (9), form (10) shows the loop refinements out of order to save space and make (10) easier to understand. The range of j is now [1,(n-2)] rather than [0,(n-1)] because of the effect of the partitioning condition.

4) Inlining Intermediate Language Definitions

The DSLGen™ Inlining phase will inline the IL definitions, replacing the convolution expressions with their definitions such as that of the convolution operator, i.e.,

```
(* (a[(row sp-Edge1 a[i,j] p q),  (col sp-Edge1 a[i,j] p q)] )
   (w sp-Edge1 a[i,j] p q))^2                                 (11)
```

for each partition specific INS clone. The inlining will continue recursively for the lower level IL definitions, e.g., row, col and w (where row and col map from neighborhood coordinates to matrix coordinates). Since (w sp-Edge1 a[i,j] p q) is defined as 0 in (5), expression (11) partially evaluates to 0. Similarly, all other convolution expressions involving edges partially evaluate to 0. After all of the inlining and partial evaluation (but before adding local declarations), expression (9) becomes expression (12):

```
void Sobel Edges9( )
      { /* Edge1 partitioning condition is  (i=0) */
        {for (int j=0; j<=(n-1);++j) b [0,j]=0;}
       /* Edge2 partitioning condition is  (j=0) */
       {for (int i=0; i<=(m-1);++i) b [i,0]= b [0,j]=0;}
       /* Edge3 partitioning condition is  (i=(m-1)) */
       {for (int j=0; j<=(n-1);++j) b [(m-1),j]=0;}
       /* Edge4 partitioning condition is  (i=(n-1)) */
       {for (int i=0; i<=(m-1);++i) b [i, (n-1)]= 0;}
       _endthread( ); }                                       (12)
```

And analogously for the **?DoASlice** routine, after a series of inlining steps analogous to the Edge1 partition refinement process but without the extensive simplification engendered by the IL definitions for the edge partitions, the center slice partition case (10) refines into:

```
void SobelCenterSlice10 (int h)
    {long ANS45;  long ANS46;
     /*  Center5-KSegs partitioning condition is
         (and (not (i=0))  (not (j=0))  (not (i=(m-1)))
              (not (j=(n-1)))) */
      /*  Center5-ASeg partitioning condition is
          (and (not (i=0))  (not (j=0))  (not (i=(m-1)))
               (not (j=(n-1))) (h<=i)(i<=(min (h+4),(m-1))))*/
       for (int i=h; i<min((h+ 4),(m-1)); ++i) {
         for (int j=1; j<=(n-2); ++j) {
           ANS45 = 0;
           ANS46 = 0;
           for (int p=0; p<=2; ++p) {
             for (int q=0; q<=2; ++q) {
               ANS45 +=
                (((*((*(b + ((i + (p + -1))))) + (j + (q + -1)))))*
                  ((((p - 1) != 0) && ((q -  1) != 0)) ? (p - 1):
                    ((((p - 1) != 0) && ((q -  1) == 0)) ?
                      (2 * (p - 1)): 0)));
               ANS46 +=
                (((*((*(b + ((i + (p + -1))))) + (j + (q + -1)))))*
                  ((((p - 1) != 0) && ((q -  1) != 0)) ? (q - 1):
                    ((((p - 1) == 0) && ((q -  1) != 0)) ?
                      (2 * (q -  1)): 0)));   }}
          int i1 = ISQRT ((pow ((ANS46), 2) +
                           pow ((ANS45), 2)));
           i1 = (i1 < 0) ? 0 : ((i1 > 0xFFFF) ? 0xFFFF : i1);
           ((*((*(A + (i))) + j))) = (BWPIXEL)  i1;   }}
            _endthread( ); }                                          (13)
```

The examples are adapted from generated code to accommodate the format and space available. For example, in generated code, i and j would be generated names like idx3 and idx4. Similarly, p and q would be something like p15 and q16. Additionally, in (13), a discussion of the introduction of the answer variables (e.g., ANS45) and the masking expression near the end is beyond the scope of this paper.

The reader will note that the inlining step has introduced some common sub-expressions (e.g., (p - 1)) which will degrade the overall performance if not removed. If this code is targeted to a good optimizing compiler, these common sub-expressions will be removed by that compiler and thereby the performance improved. However, if the target compiler is not able to perform this task, DSLGen™ offers the option of having the generator system remove the common sub-expressions and this can be easily added to the specification of the execution platform. However, the common sub-expressions are explicitly included in this example (i.e., not optimized away) to make the connection to the structures of the MTs used by the INS more obvious to the reader. The broad structure of the right hand operand of the times (*) operator in the right hand side of the assignments to the answer variables ANS45 and ANS46 is structurally the same as that of the W method transform specialized to the center partition for SP and S. That is, the right hand side of ANS46 is the C form:

```
((((*((*(b + ((i + (p + -1))))) + (j + (q + -1)))))    *
  ((((p - 1) != 0) && ((q -  1) != 0)) ? (q - 1):
   ((((p - 1) == 0) && ((q - 1) != 0)) ? (2 * (q - 1)): 0)))   (14)
```

It mimics the form of the MT definition for w of SP-Center5 because (14) is derived by inlining that MT definition and eventually processing it into legal C. For reference, the rhs of the MT definition of w of SP-Center5 has the form

```
(if (and (!= ?p 0) (!= ?q 0))
    (then ?q)
    (else (if  (and (== ?p 0) (!= ?q 0))
               (then (* 2 ?q))
               (else 0)))).                                           (15)
```

When the inlining occurs, the SP-Center5 generator pattern variable ?p is bound to "(p - 1)" and ?q is bound to "(q - 1)". The "- 1" part of these values arise because of the C indexing design feature encapsulated earlier in the generation process. Recall that that design feature maps the domain language indexing system for neighborhoods (i.e., [-n, +n]) to a C language style of indexing (i.e., [0, 2n]).

VI. THE DSLGEN™ PROTOTYPE

DSLGen™ is the culmination of a multi-year R&D effort and comprises about 52KLOC of CommonLisp and CLOS running on Franz Allegro, version 8.2. A key component of the architecture upon which many of the other components are built is a general pattern matching system with backtracking, which is built using continuations. Built on top of the pattern matcher is a transformation system that includes several flavors of transformations (e.g., general, MTs, generic components, and deferred, where deferred transformations are used to move newly created subtrees up the abstract syntax tree). The transformations are specific to a specific generator phase (i.e., they are only enabled during the named generator phases to which they apply, e.g., the SyntheticDesign phase).

The partial evaluator and several specialized inference subsystems are also heavy users of the pattern matcher. The inference systems include a type inference system, a backchaining rule system, a logical expression simplifier based on logical subsumption and an inequality inference engine based on Fourier-Motzkin elimination. The inequality inference engine is used for inferring relationships among logical architecture elements (e.g., inferring that (i == (m - 1)) is false when (i == 0) and (m>1) are true) .

As to generation performance, a Sobel example on RGB images with partitioning but without load leveling or threads takes about 75 seconds to generate an Abstract Syntax Tree (AST) for C (or 40 to 50 seconds if not generating history and traces). Adding in the surface syntax to generate the text-based C files adds an additional 15 to 20 seconds. Adding multicore with threads, SIMD (Single Instruction, Multiple Data) instructions and various other architectural complexities increases generation times by small amounts.

VII. PERFORMANCE TESTING

Generated code was tested with a selection of implementation variations on a 4 core, 3.33 GHz Velocity brand computer with 12 GB of real and 24 GB of virtual memory. The test computer is built on the Intel i7 CPU with Turbo mode, which allows overclocking when the CPU is running under maximum temperature and power specification. It has 8 virtual processors. The code was compiled with Microsoft's Visual Studio 2008 C/C++ compiler.

The test data was a 215 by 215 pixel image in RGB format with a 24 bit pixel depth. The chosen computations included Sobel and Wallis edge detection methods [28] since they put a greater computational load on the machine than other possible computations might. In addition, Sobel provides one of the more serious challenges to the generator in that it requires use of virtually all of the generation facilities. The testing also included image Average and Unsharp Mask [28] (often used to sharpen Mammogram images), both of which have lighter computational loads.

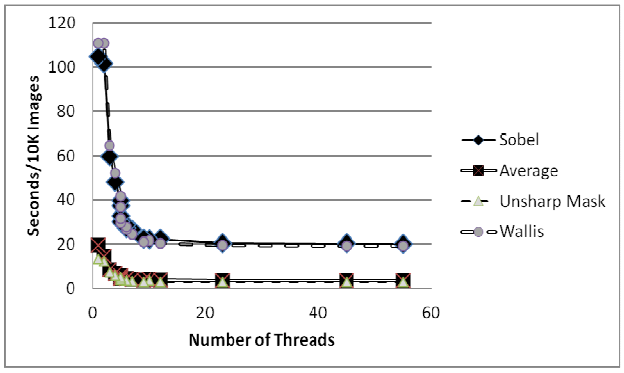

Figure 6. Performance vs. thread count

Figure 6 shows the results for various computations decomposed into threads to be run in parallel. These tests were run 10,000 times per image. For Sobel, the best performance was achieved at 55 threads, which required approximately 20.3 seconds to run the full set, or about 2 milliseconds per image. The worst results were with two threads, one for the edge cases and one for the center, which required approximately 105 seconds for the 10,000 images or a bit over a 10 millisecond per image. This was roughly the same time required for the calibration case, a hand coded version compiled to use no parallelism of any kind. Notice that the time drops quickly with five threads (i.e., one for the edges and four for the image center), taking about 32.8 seconds for the full set of images or about 3.3 milliseconds per image. This is about what simple logic would expect with four cores. However, the time continues to improve modestly for each five or so additional threads until it begins to level out at about 20.5 seconds at about 23 threads. Thereafter, the improvement is a tenth of a second or so for five or so additional threads. It is somewhat counter intuitive that one should get any improvement at all after the image has been evenly decomposed over the four cores. It is not entirely clear why this occurs but our current hypothesis is that it may be the "GPU effect" where many threads can mask memory, cache or other kind of latency if thread switching is efficient enough. Also, fast thread switching among virtual processors in the hardware (called Hyperthreading) may play a role. The target computer has two virtual processors per core and this is known to increase overall performance in many cases.

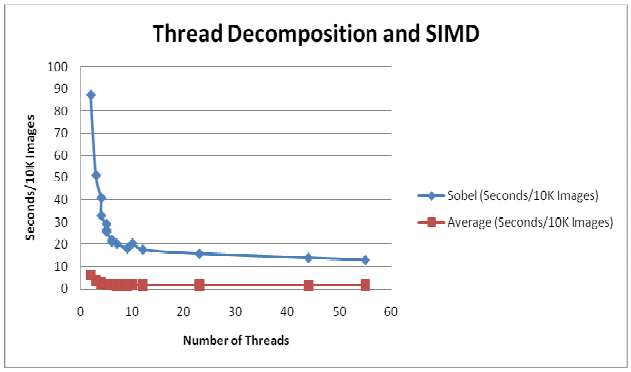

Figure 7. Threads and SIMD

Other test cases with different kinds of image processing functions show similar behavior although the computational loads vary based on the nature of the computation. Sobel and Wallis have computational heavy loads that just simply require hefty computational capacity. Sobel employs square roots and Wallis uses logarithms. On the other hand, Average and Unsharp Mask are both light weight computations that employ little more than addition and division. Hence, they require less computational capacity as is clear from the graph.

With the addition of SIMD instructions (Fig. 7), the added improvement for Sobel ranges from about a 14% improvement for few or no threads to 36% for the maximum number of threads tested. With only two threads, Sobel took 87.2 seconds for all 10K images or 8.7 milliseconds per image, whereas with 55 threads, it took 12.87 seconds for all 10K images or about 1.3 milliseconds per image.

Interestingly, the SIMD effect with threads for Image Average was significant and somewhat surprising – between 57% and 58% improvement regardless of the number of threads. What this suggests is that virtually all of the computation for Image Average can be done with instruction level parallelism (i.e., addition of a vector of numbers) leaving only a single additional arithmetic operation (i.e., multiplication or division by a constant) to be done via the standard arithmetic unit. So, the multicore parallelism improvement is a much flatter curve. Once one runs out of processors there is little additional effect although there may be some additional effect depending on the structure of the computation and its possible effect of processor latencies. Recall the comments from the previous discussion about the counter intuitive improvement for Sobel beyond 5 or so threads.

By contrast, a much smaller amount of the Sobel computation is done in parallel instructions (i.e., the PMADD operation on the three weight vectors and the three corresponding pixel vectors, where each of the vectors is only three numbers long) whereas the addition of the intermediate instructions (via the PADD instruction) is relatively inconsequential. But more importantly, the square, addition and square root operations that follow the SIMD instructions will be done on the standard CPU arithmetic unit. These last three operations (and especially the square root operation) are likely to be the lion's portion of the computation. Therefore, the really big improvement with Sobel arises because of the large scale parallelism provided by the thread based parallelism and this tends to swamp much of the savings of the instruction level parallelism of the SIMD instructions.

VIII. Related Research

A key difference between most previous research and DSLGen™ is that DSLGen™ starts working strictly in the problem domain and programming process domain rather than the PL domain. Virtually all previous research chooses representation systems that are based to some degree upon PL constructs or abstractions thereof. This includes compiling technology, generator technology [4] [5] [7] [22] [29] [30], computer aided software engineering (CASE) [13], model driven engineering [22], Aspect Oriented Programming (AOP) [17], Anticipatory Optimization Generation (AOG) [6] [7], general optimization based methods [1] [20] [16], parallel or specialty programming languages [8] [12], programming languages superficially similar to DSLGen™'s partitioning model [14], programming language augmentation systems [15] [27], maintenance support systems [1] [2], refactoring [18] and other related technology and methods for creating implementation code from a specification of a computation. The PL based representational choice forces conventional generation technologies to introduce design and PL forms, implementation structures, organizational commitments and other execution platform based details too early and thereby make design decisions about the architecture of the solution that will prevent other desired design decisions from being made later. Or at least, it will make those other desired design decisions require revision of the model or design, which is very often difficult to automate within a PL oriented domain.

In general, there are two important properties that differentiate these various approaches from DSLGen™: 1) The specifications of the computations in these approaches are typically not invariant over all or even a variety of execution platform architectures, and 2) target program implementations exploiting specific high capability features cannot be fully and automatically generated without compromising the invariance property. That is, user action is required either to revise the computational specification model to fit the new execution platform or to extend an overly abstract and therefore incomplete input specification to target a specific execution platform. Generation of target program implementations for a variety of execution platforms that exploit the execution platform features (e.g., multicore parallelism, vector instructions, etc.) requires human redesign or reprogramming in one form or another. For example, in these approaches, the transition from one execution architecture (e.g., simple Von Neumann) to another (e.g., multicore and/or vector machines) requires user action to adapt the computation specification or model to the new execution architecture.

In many cases, these conventional technologies often force a top down, reductionist approach to design where the top level programming structure and the essence of its algorithm are expressed first and then the constituent essence is recursively extended step by step until the lowest level of PL details are expressed. However, that initial structure may be incompatible with some desired design requirements or features that are addressed later in the development or generation process. The initial design may have to be reorganized to introduce such design requirements or features. For example, the requirement to fully exploit a multicore computer requires a significant, difficult and many step reorganization to fully exploit the performance improvements possible with multicore. Automation of such reorganizations at the programming language level is seriously complicated and except for relatively simple cases is prone to failure. This is why compilers that can compile programs written and optimized for one execution platform are often unable to satisfactorily compile the same programs for a different execution platform with an architecture that employs a significantly different model for high capability execution and fully exploit the high capability features of the new architecture. For example, programs written for the pre-2000 era Intel platforms are largely unable to be automatically translated to fully exploit the multicore parallelism of the more recent Intel platforms. Human based reprogramming is almost always necessary to fully exploit the multicore parallelism.

While much research has been highly PL oriented, some research is clearly working in the problem domain. A prime example is the work of Jim Neighbors, who introduced the idea of using domain specific information in program generation. [24] [25] [26] His approach is to map from purely problem domain oriented languages through a series of language to language mappings, incrementally evolving to pure programming language representations. While DSLGen™ is consistent with that spirit, the underlying machinery (e.g., the non-reductionist design approach, the non-PL logical architecture model, the APCs, the incremental design feature encapsulation and the incremental addition of sets of PL features phase by phase) distinguishes the DSLGen™ approach from Neighbor's work. Nevertheless, Neighbors' work has made significant contributions to program generation from which this work has benefited.

IX. Contributions

The contributions of this work are due in large part to the fact that this work breaks with convention in a number of ways. Perhaps the most important break is avoiding the PL domain in the initial modeling process. This allows the

implementation neutrality of the ***INS*** and allows ***the separation of the INS from the specification of the execution platform (EXPS)*** while still allowing the generated programs to ***exploit the full range of high capability features*** of the EXPS platform. While some systems emphasize language neutrality [30] rather than implementation neutrality, their specifications clearly derive from the PL domain and they therefore inherit the liabilities of the PL domain.

The ability of DSLGen™ to exploit high capability features arises from another important contribution, specifically, the design representation system based on ***associative programming constraints***. The design representation system allows the initial and early stage designs to be organized as ***logical architecture***s thereby allowing the system to ***operate in the problem and programming process domains*** and to introduce PL constructions and assumptions incrementally. Operating with ***problem domain concepts*** such as edge and center partitions allows DSLGen™ to begin to manipulate and extend the LA without (initially) being restricted by the constraints inherent to programming languages.

Organizing the ***IL definitions*** as ***provisional transformations that are malleable*** provides the opportunity for ***incrementally adding design features*** by using higher order transformations to revise the IL definitions to incorporate those features but still defer casting them into programming language constructs until late in the generation process. Thus, the IL becomes the stand-in or precursor representation for the code details that have yet to be concretely determined. For example, expressions like Partestx(sp) can stand-in for code or meta-information (e.g., assertions) that cannot be refined to concrete form until the implementation context (e.g., a specific partition) and locale (i.e., the location in the AST) are concretely and finally determined. And when that context is eventually pinned down (e.g., to Edge1), Partestx(sp) can be specialized (e.g., to Partestx(sp-Edge1)), which will move it a step closer to refinement into a concrete logical expression.

DSLGen™ relies heavily on ***inference and implication***. For example, the APCs are described by a set of logical assertions that are augmented as the design progresses. This allows ***architectural features and programming clichés to be expressed inferentially*** rather than structurally and proscriptively. This defers making PL level design decisions. These PL representational forms are hard to revise, change and manipulate. For example, in DSLGen™, adding messy design details and programming clichés can be deferred until the broad architectural structure is settled.

In summary, DSLGen™ represents a fundamentally new paradigm for program generation.

X. ACKNOWLEDGMENTS

I want to thank two contributors who have been helping with the commercialization of DSLGen™ – Mitch Lubars and Rob Pettengill. Mitch did the performance testing and implemented the Fourier-Motzkin based inference engine. Additionally, Mitch wrote a prototype of a Slicer/Slicee target program from which the author abstracted the Slicer/Slicee design pattern used to support one class of synthetic partitions. Rob built a search and bookmarking facility for the transformation history debugger. Additionally, both Mitch and Rob read an early version of this paper and provided many comments and suggestions that improved it.

References

[1] Robert L. Akers, Ira D. Baxter, Michael Mehlich, Brian J. Ellis, and Kenn R. Luecke, "Case study: Re-engineering C++ component models via automatic program transformation," Information and Software Technology 49, 2007, pp. 275-291.

[2] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich, "DMS®: Program Transformations for practical scalable software evolution," International Conference of Software Engineering, May 2004, pp. 10.

[3] David F. Bacon, Susan L. Graham, and Oliver J. Sharp, "Compiler transformations for high-performance computing," ACM Surveys, Vol. 26, No. 4, December, 1994, pp. 345-420.

[4] Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas, "Scalable software libraries," Symposium on the Foundations of Software Engineering. Los Angeles, California, 1993, pp. 191-199.

[5] Ted J. Biggerstaff, "A perspective of generative reuse, annals of software engineering," Baltzer Science Publishers, AE Bussum, The Netherlands, 1998, pp.169-226.

[6] Ted J. Biggerstaff, "Fixing some transformation problems" Automated Software Engineering Conference, Cocoa Beach, Florida, 1999, pp. 10.

[7] Ted J. Biggerstaff, "A new architecture of transformation-based generators," IEEE Transactions on Software Engineering, Vol. 30, No. 12, Dec., 2004, 1036-1054.

[8] Ted J. Biggerstaff, "Automated partitioning of a computation for parallel or other high capability architecture," Patent no. 8,060,857, United States Patent and Trademark Office, filed January 31, 2009, issued November 15, 2011.

[9] Ted J. Biggerstaff, "Non-localized constraints for automated program generation," United States Patent and Trademark Office, Patent no. 8,225,277, filed April 25, 2010, issued July 17, 2012.

[10] Ted J. Biggerstaff, "Synthetic partitioning for imposing implementation design patterns onto logical architectures of compuatations," United States Patent and Trademark Office, Patent no. 8,327,321, filed August 27, 2011, issued Dec. 4, 2012.

[11] Guy E.Blelloch, Jonathan C.Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha, "Implementation of a portable nested data-parallel language," in Proceedings of PPOPP '93 Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming, 1993, 102-111

[12] Guy Blelloch, "Programming parallel algorithms," Communications of the ACM, 39 (3), March, 1996, pp. 85-97.

[13] CASE Tools, See http://en.wikipedia.org/wiki/Rational_Rose .

[14] Bradford L. Chamberlain, Choi, Sung-Eun, Deitz, Steven J. and Snyder, Lawrence, "The high-level parallel language ZPL improves productivity and performance," Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing, 2004, pp. 1-10.

[15] Barbara Chapman, Gabriele Jost and Ruud Van Der Pas, Using OpenMP, MIT Press, 2008.

[16] Daniel E.Cooke, J. Nelson Rushton, Brad Nemanich, Robert G.Watson, and Per Andersen, "Normalize, transpose, and distribute: an automatic approach to handling nonscalars," ACM Transactions on Programming Languages and Systems, Vol. 30, No. 2, 2008, pp. 49.

[17] Tzilla Elrad, Robert E. Filman and Atef Bader, "Aspect-oriented programming," Communications of the ACM, Vol. 44, No. 10, 2001, pp. 29-32.

[18] Martin Fowler, Kent Beck, John Brant and William Opdyke, "Improving the design of existing code by refactoring," Addison-Wesley, 2000, pp 431.

[19] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Design Patterns, Addison Wesley, 1995.

[20] M. W. Hall, S. P.Amarasinghe, B. R.Murphy, S. W.Liao, and M. S.Lam, "Interprocedural parallelization analysis in SUIF," ACM Transactions on Programming Languages and Systems, Vol. 27, No. 4, July, 2005, pp. 662-731.

[21] Neil D.Jones, "An introduction to partial evaluation," ACM Computing Surveys, Vol. 28, No. 3, 1996, pp. 480-503.

[22] Steve Macdonald, Kai Tan, Jonathan Schaeffer, and Duane Szafron, "Deferring design pattern decisions and automating structural pattern changes using a design-pattern-based programming system," ACM Transactions on Programming Languages and Systems, Vol 31, No. 3, April, 2009.

[23] Model Driven Engineering. See http://en.wikipedia.org/wiki/Model-driven_engineering .

[24] James M. Neighbors, "The Draco approach to constructing software from reusable components," IEEE Transactions on Software Engineering, SE-10 (5), (Sept. 1984) pp 564-573.

[25] James M. Neighbors, "Draco: a method for engineering reusable software systems," In: Ted Biggerstaff and Alan Perlis (eds.): Software Reusability, Addison-Wesley/ACM Press (1989), pp. 295-319

[26] James M. Neighbors, see http://www.bayfronttechnologies.com/.

[27] OpenMP Architecture Review Board, "OpenMP Application Program Interface," Version 3.0, May 2008.

[28] Gerhard X. Ritter and Joseph N. Wilson, The Handbook of Computer Vision Algorithms in Image Algebra, CRC Press, 1996.

[29] Kai Tan, Duane Szafron, Jonathan Schaeffer, John Anvik And Steve Macdonald, "Using generative design patterns to generate parallel code for a distributed memory environment," Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming, June, 2003, pp. 203-215.

[30] Satnam Singh, "Computing without processors," CACM, Sept. 2011, pp. 46-54.

[31] Unified Modeling Language. See http://en.wikipedia.org/wiki/Unified_Modeling_Language.