Taming Multi-core Parallelism with Concurrent Mixin Layers

Jeremy Archuleta Tom Scogland Eli Tilevich

Department of Computer Science
Virginia Tech
Blacksburg, Virginia, 24061
{jsarch, njustn, tilevich}@cs.vt.edu

Abstract

The recent shift in computer system design to multi-core technology requires that the developer leverage explicit parallel programming techniques in order to utilize available performance. Nevertheless, developing the requisite parallel applications remains a prohibitively-difficult undertaking, particularly for the general programmer. To mitigate many of the challenges in creating concurrent software, this paper introduces a new parallel programming methodology that leverages feature-oriented programming (FOP) to logically decompose a product line architecture (PLA) into concurrent execution units. In addition, our efficient implementation of this methodology, that we call *concurrent mixin layers*, uses a layered architecture to facilitate the development of parallel applications. To validate our methodology and accompanying implementation, we present a case study of a product line of multimedia applications deployed within a typical multi-core environment. Our performance results demonstrate that a product line can be effectively transformed into parallel applications capable of utilizing multiple cores, thus improving performance. Furthermore, concurrent mixin layers significantly reduces the complexity of parallel programming by eliminating the need for the programmer to introduce explicit low-level concurrency control. Our initial experience gives us reason to believe that concurrent mixin layers is a promising technique for taming parallelism in multi-core environments.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming; D.2.2 [*Design Tools and Techniques*]: Modules and interfaces; D.2.10 [*Design*]: Methodologies; D.2.11 [*Software Architectures*]: Data abstraction, Patterns

General Terms Design, Experimentation, Languages

Keywords multi-core, concurrency, compositional programming, product lines, features

1. Introduction

Due to the recent shift in computer hardware design, parallel programming has entered the mainstream. Today, new processors combine multiple cores on the same chip, with each new generation of processors featuring more cores. As a result, computing applications must use multiple cores in parallel to realize available performance.

While explicit parallelism has long been a mainstay in specialized domains such as high-end scientific computing, most general-purpose software is written without parallelism in mind. Thus, these applications will have to embrace parallelism to maintain performance as computer hardware evolves.

Approaches to creating parallel programs range from fully automated to fully manual. Fully automated approaches such as parallelizing compilers can improve performance to a certain extent, at which point the programmer will have to provide an alternative algorithm to realize further improvements. Thus, a fully automatated approach may be unable to achieve the desired level of performance without programmer intervention.

At the other extreme, creating parallel applications manually through low-level concurrency mechanisms remains a challenging and involved endeavor, requiring special expertise and attention to detail to ensure both correctness and performance. In particular, using existing threading and OS synchronization primitives requires detailed and extensive analysis to safely improve performance. Therefore, using low-level concurrency mechanisms may not be an appropriate means for the general programmer to create parallel programs.

Several technologies have explored the middle ground between automatic and manual (15; 16; 18; 30). However, in our view, these technologies do not focus on serving the needs of the general programmer, who would:

- be an expert in their application domain, but not necessarily in parallel programming;
- aim to achieve reasonable performance improvements rather than optimal performance;
- prefer an intuitive and usable solution that does not require low-level systems programming;
- want to leverage their expertise in a mainstream programming language and its standard features and libraries.

In light of these observations, this paper presents concurrent mixin layers, our approach to creating parallel programs, which focuses on the general programmer by embracing the methodologies of Product Line Architectures (PLA) and feature-oriented programming (FOP) to create parallel applications capable of utilizing multiple cores. Using FOP, concurrent mixin layers separates the concern of parallel program design from its implementation mechanism by identifying concurrency at a high level. Furthermore, it builds upon the success of layered architectures as a facility for functional decomposition.

Specifically, our approach uses the mixin layers layered architecture as a mechanism for mapping program features into concurrent units. As with traditional mixin layers, the programmer expresses program features as separate layers. However, concurrent mixin layers also enables the programmer to express which features

should be executed in parallel and uses generative techniques to transparently introduce the necessary low-level concurrency mechanisms. Following the active object pattern (26) concurrent mixin layers extracts parallelism at the method level using futures (21).

Accordingly, this paper makes the following contributions:

- A novel parallel programming methodology that uses PLA and FOP to separate concurrency concerns from the core functionality of a program.
- A new approach for creating parallel programs, concurrent mixin layers, that can introduce concurrency at multiple levels of granularity.
- An efficient and reusable implementation of concurrent mixin layers, capable of leveraging multiple cores for improved performance.

To support these claims, the rest of this paper is structured as follows. Section 2 provides a technical background for this work. Section 3 then presents our programming methodology for creating parallel applications with Section 4 detailing our reference implementation in C++. To validate our methodology and implementation, we provide a case study of using concurrent mixin layers in Section 5 followed by a discussion on the state of the art in software support for parallel programming on multi-core systems in Section 6. Finally, Section 7 outlines future work directions, and Section 8 offers concluding remarks.

2. Background

For decades, computer hardware manufacturers have improved processor performance through increases in clock frequency. However, while the number of transistors on a processor continues to double approximately every 18 months, familiarly known as Moore's Law, processors have hit a "power wall" where clock frequency will not significantly increase without "heroic measures" (29). Thus, to provide more computing capability on a single processor, hardware manufacturers have turned to multi-core technology.

The principle behind multi-core technology is to provide multiple, *simplified* computing cores on a processor rather than a single *complex* computing core (Figure 1). To this end, a single processor with two cores can provide twice the computing capability of a single core operating at the same clock frequency; four cores can provide four times the capability. For hardware manufacturers, multi-core technology is a "saving grace", as it allows them to continue doubling the capable performance of the processor by doubling the number of cores.

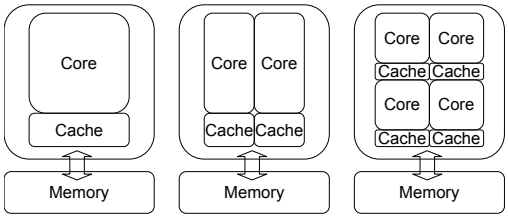


Figure 1. Examples of Single-, Dual-, and Quad-core Systems

For the programmer, on the other hand, the transition to multi-core technology signifies that "the free lunch is over." The performance of an application no longer automatically doubles every 18 months because it can run on a processor with twice the clock frequency; instead the programmer has to modify the application to utilize twice as many cores. Furthermore, the outlook for multi-core technology is "many-core" technology: hundreds of energy-efficient (i.e., lower-frequency) cores (Figure 2).

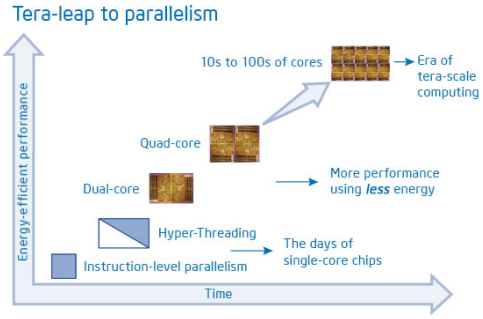


Figure 2. Intel's Outlook of Many-core Technology (http://www.intel.com)

This shift to multi-core technology requires that the programmer embrace concurrency to scale the performance of their software with the potential of the hardware – by no means an easy task. However, this process can be facilitated through novel software engineering solutions focused on improving parallel programming development. To this end, this paper presents a novel parallel programming methodology that leverages feature-oriented programming to logically decompose a product line architecture into concurrent execution units. We introduce the primary building blocks of this parallel programming methodology next.

2.1 Product-Line Architectures

A Product-Line Architecture (PLA) is a methodology for creating a family of related software applications (32). The intuition behind a PLA is that organizations tend to release software applications that are closely related (i.e., belonging to the same product-line). By enabling the sharing of software components among the related applications, PLAs reduce the overall complexity of software development and maintenance. While several methodologies have been proposed for building and analyzing PLAs (14; 32; 37), this work uses feature-oriented programming to apply object-oriented methodologies to the creation and maintenance of product-lines.

2.2 Feature-Oriented Programming

Feature-oriented programming (FOP) is a software development methodology in which features are first-class citizens in the software architecture (33). That is, FOP decomposes applications into a set of features that together provide the requisite functionality. Composing multiple objects from a single set of features separates the core functionality of an object from its refinements making the resulting software more reusable and robust.

In addition, FOP allows easy mix-and-match composition of features in a modular fashion, as an application is built using stepwise refinement. A common implementation strategy in FOP is to use a layered architecture, in which layers correspond to features. The resulting "feature stack" is composed of many layers with each layer 1) providing a single feature, and 2) refining existing features in the stack (4; 5). As an example, Figure 3 illustrates how a set of four features can be composed to create a variety of objects.

2.3 Mixins and Mixin Layers

Both mixins and mixin layers are well-suited for implementing FOP designs, as both naturally enable incremental refinement through their layered architecture.

Specifically, a mixin is an abstract subclass that can be applied to different superclasses to specialize the behavior of a family of classes (10). This mechanism is similar to that of subtype inheritance, whereby subclasses specialize the behavior of their superclasses. However, unlike subtype inheritance, mixins build

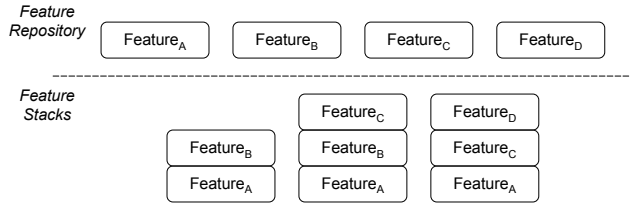
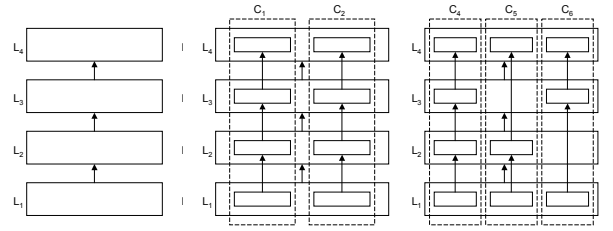


Figure 3. Composition of a Set of Features into Feature Stacks

the inheritance tree bottom-up, rather than top-down (Figure 4(a)). For example, even though `Cat < Animal > mixinAnimalCat` and `Cat < Picture> mixinPictureCat` share the same mixin subclass, it is the superclass (i.e., `Animal` or `Picture`) that determines the functionality.

Mixin layers is an approach to implementing collaboration-based designs. In C++, mixin layers use inner classes to model different collaborating roles within each layer (34). The enclosing classes and their inner classes form inheritance relationships with the corresponding classes in the layer above (Figure 4(b)). Mixin layers enable the programmer to add functionality in a flexible, and yet well-defined manner: each added layer contains only the inner classes needed to provide the required functionality (Figure 4(c)).



(a) Stack of mixins (b) Stack of Mixin Layers (c) Alternative Stack of Mixin Layers

Figure 4. Inheritance Hierarchies of Mixins and Mixin Layers

We have chosen mixin layers as the architecture for this work, as it allows each feature to be a collaboration of smaller, encapsulated units, while still providing step-wise refinement.

3. Methodology: Features as Units of Concurrency

Any acceptable parallel programming methodology for the general programmer needs to satisfy the requirements of both usability and performance. A methodology that fails to satisfy both objectives is not likely to become widely applicable, but could still find use in highly-specialized domains such as high-end scientific computing. For mainstream computing, we operate under the assumption that improved usability at the expense of a slight deterioration in performance is a reasonable trade-off.

Specifically, a parallel programming methodology should deliver the following benefits to the general programmer:

- intuitive parallel programming abstractions at an appropriate level to separate high-level design from low-level concurrency details
- ease of identifying concurrency in the high-level design
- improved extensibility and maintainability as the design evolves

With these requirements in mind, we chose to use FOP as an enabling technology for parallel programming as detailed below.

3.1 Appropriate Level of Parallel Programming Abstractions

Recall that FOP is a methodology, whereby the programmer reasons about an application in terms of its features. Correspondingly, individual features are conceptual facets of a program, and as such may not necessarily correspond to specific language constructs such as classes, methods, and fields. As such, we believe FOP provides an appropriate level of parallel programming abstraction for the general programmer, as features, and not parallel programming technologies or languages, are the units of consideration.

When using FOP to introduce parallelism, a clear separation between the high-level design units and low-level implementation artifacts exists, making it possible to abstract away the specific details of the underlying parallel programming technology. This allows FOP to reduce the burden of developing a parallel application by eliminating the need to master any specific parallel programming technology when designing programs with parallel execution in mind. That is, a feature-oriented design can hide the implementation of parallelism, whether it is PThreads(19), MPI(18), or the parallel programming technology du jour, from the developer, thereby bridging the gap between the mainstream programming technologies of today and the specialized parallel programming technologies of tomorrow.

3.2 Identification of Concurrency

In this era of multi-core technology, in which performance is gained by increasing the number of cores, it is imperative that software exhibit sufficient levels of parallelism. The stepwise refinement approach of adding features to an application as the software evolves provides an avenue to identify and extract the requisite parallelism. To highlight the opportunities afforded by the use of FOP as a means of introducing concurrency, we distinguish between stepwise refinement of the *application* and stepwise refinement of the *features* in the context of PLAs.

When new features are added to refine an application, the programmer can easily choose to execute these new features as concurrent units because of the inherent low coupling between offered by these features. For example, in Figure 5 the `building` product line features: `plumbing`, `electrical`, `mobility`, and `architectural` are prime choices for encapsulation into concurrent units as they exhibit low-coupling between them.

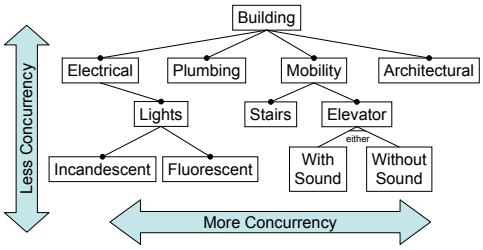


Figure 5. Identifying Concurrency within a Product Line

On the other hand, when features refine existing features, a higher level of coupling exists limiting the amount of concurrency available. However, concurrency may still be introduced, albeit to a lesser degree. For example, `with sound` (e.g., an elevator that plays music) refines an `elevator`, thereby exhibiting a higher degree of coupling. However, while `elevator` uses the feature `with sound` to play music (i.e., a less concurrent operation), the `elevator` does not need to use this feature to move between floors (i.e., a more concurrent operation).

It is worth noting that features with minimal coupling, such as `stairs` and `elevator`, can be nearly-completely concurrent.

In summary, by analyzing the relationships between features within an FOP design, various levels of concurrency can be identi-

fied. Furthermore, as more features are added, more opportunities for concurrency are created. That is, as the software matures and evolves, an FOP design makes it possible to use increasingly-larger numbers of cores.

3.3 Extensibility and Maintainability

The maintainability and extensibility advantages offered by FOP and PLA have been thoroughly documented in the literature (9; 8; 36). The advantages that matter most to parallel programming include the following:

- **Improved modularity.** Independent and encapsulated modules readily offer possible delineations of concurrent units.
- **Higher-level abstractions.** Features separate the design concepts from the implementation details, thereby allowing the programmer to focus on the algorithmic correctness of a parallel application.
- **Incremental refinement.** Adding and removing features in a defined and incremental manner provides a systematic method for debugging a parallel application.

Collectively, these features help to provide a bridge from single-threaded applications to multi-core parallel programs for the general programmer due to their far-reaching affects during the development, deployment, and maintenance processes.

3.4 Range of Applicability

Even with the aforementioned benefits of using FOP as a parallel programming methodology, it is important to recognize its range of applicability to the problem of creating applications for multi-core systems.

3.4.1 Legacy Applications

Despite the benefits that FOP and PLAs offer to the general programmer, many applications are not constructed using these technologies. Therefore, these applications cannot benefit from the advantages of our approach. Fortunately, automated tools to transition legacy applications into feature-oriented designs have been investigated (24; 27).

3.4.2 Amdahl's Law

Parallel performance on multi-core systems is commonly described by Amdahl's Law (3):

$$S = \frac{1}{1 - C} \tag{1}$$

where S is the speed-up of the application and C is the portion of the application that is concurrent. When an application is only 10% non-concurrent (i.e., 90% concurrent), the application can only achieve a *maximum* speedup of $10x$. Therefore, to achieve optimum performance a program built using FOP requires that the product line have as many lowly-coupled, highly-concurrent features as possible.

As a consequence, identifying concurrency only via FOP may not be suitable for high-end computing experts seeking to obtain optimal performance. However, FOP provides a powerful tool for general programmers wishing to obtain reasonable performance improvements in a maintainable and extensible fashion without the requirement of first understanding low-level concurrency details of any specific parallel programming technology.

4. Implementation: Concurrent Mixin Layers

To verify the concept of using FOP as a parallel programming methodology, we have implemented a novel application of mixin layers that combines a mixin layers architecture with proven concurrent programming technologies. The implementation, that we call *concurrent mixin layers* , enables a general programmer to create specialized parallel applications without the need to understand low-level concurrency details.

The process of using concurrent mixin layers is shown in Figure 6. First, a programmer starts with their PLA viewed as an FOP design. Second, each feature is created using the standard abstractions of mixin layers and assembled into a feature stack. Then concurrency is introduced on an "as needed" basis by utilizing the programming abstractions of concurrent mixin layers with a generator adding the final "glue-code" to create a parallel application. That is, to transform a stack of mixin layers into a parallel application capable of executing on a multi-core system, the programmer simply declares which layers should be executed as concurrent units and generates the requisite intercepting methods. A key feature of concurrent mixin layers, is that the entire process can shield the programmer from explicitly having to interact directly with low-level concurrency constructs in order to introduce parallelism into a product line.

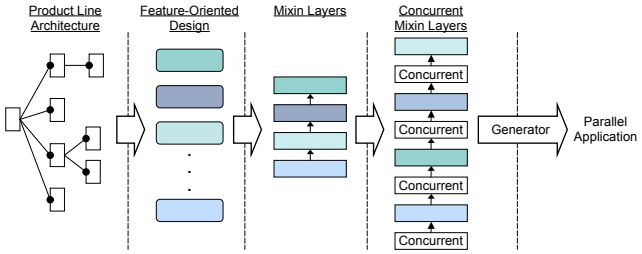


Figure 6. Progression from a PLA to a Parallel Application

As a concrete implementation strategy, concurrent mixin layers follows the active object pattern (26) by introducing a new mixin layer, `Concurrent`, to encapsulate layers (i.e., features) into concurrent objects at different levels of granularity. To ensure that the original application semantics are preserved, concurrent mixin layers refines this pattern with a set of invariants. Additionally, while shielding the general from low-level concurrency details, our implementation allows expert programmers to add low-level concurrency within a layer to further improve performance.

To demonstrate *how* concurrent mixin layers can deliver these practical benefits to the programmer, we next describe our C++ implementation. It is worth noting, however, that concurrent mixin layers is language and concurrency-mechanism agnostic.

4.1 Encapsulating Concurrency

The primary means of specifying concurrent execution in concurrent mixin layers is a special layer, called `Concurrent`, which encapsulates other layers into a single concurrent execution unit. Furthermore, `Concurrent` hides the specific low-level concurrency mechanisms from the programmer's view, effectively separating the implementation of concurrency from the design of the application. To enforce the requirement that `Concurrent` be a completely encapsulated unit of execution, private inheritance is used to form a "barrier" between the layers above and below itself. Thus, `Concurrent` requires that all method calls passing through it are intercepted.

The interception of method calls is necessary when executing layers within different execution contexts. When a layer attempts to invoke a method in a different concurrent execution unit, the invocation of that method must be redirected, altering the original control flow of the program. To enable this, we employ (and generate) proxy methods – methods that interface with the underlying low-level concurrency mechanisms.

As a concrete example, consider Figure 7, in which object `AB` is a mixin layers hierarchy of `B < A >` allowing `A` and its associated methods to be accessible from within `B`. On the other hand, when object `ABparallel` is defined as `B < Concurrent < A > >`, `A` and `B` execute concurrently. Therefore, if the programmer tries to invoke a method in `A` directly from `B`, where `Concurrent` does not first explicitly intercept the invocation, the compiler will signal an error.

```
template < typename Super >
class Concurrent : private Super {
  public:
    /** intercept 'baz(...)' **/
    void baz(...) { /** proxy body **/ }
};

class A {
  public:
    void foo(...) { /** body **/ }
    void baz(...) { /** body **/ }
};

template < typename Super >
class B : public Super { };

int main (int argc, char** argv) {
  typedef B < A > Serial;
  Serial AB;
  AB.foo(...); /** Allowed **/

  typedef B < Concurrent < A > > Parallel;
  Parallel ABparallel;
  ABparallel.foo(...); /** Unallowed
       'Error:: void A::foo()' is inaccessible;
       'A' is not an accessible base of
       'B<Concurrent<A> >' **/
  ABparallel.baz(...); /** Allowed **/

  return 0;
}
```

Figure 7. Allowed and Unallowed Accesses of `A`

The process of creating these proxy methods by hand would be tedious and error prone. Therefore, we have automated this process using generative techniques.

4.1.1 Generating Requisite Proxy Methods

Since the generation of the proxy methods is a cross-cutting concern, we first considered using an aspect extension of C++, AspectC++ (35), which can add methods to existing classes. However, we ended up not using AspectC++, as its current version cannot weave into template classes, which C++ mixins use heavily. We also briefly considered using the aspectual mixin layers of FeatureC++ (4), but the current implementation is only available for Windows.

Our current implementation of concurrent mixin layers uses a C++ parsing utility called GCCXML (20), which taps into the platform's C++ compiler (e.g., GCC on UNIX or Visual C++ on Windows) to create a structured XML description of the user's mixin layers stack. The XML description is passed to a Perl script that generates the corresponding proxy methods within `Concurrent` creating a parallel application (Figure 8).

4.2 Creating Parallelism at Multiple Granularities

By using a mixin layer to encapsulate concurrency, concurrent mixin layers enables programmers to create parallelism at different levels of granularity. That is, concurrent mixin layers enables programmers to choose which layers to combine into concurrent

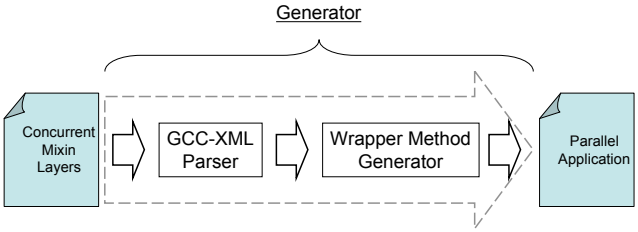


Figure 8. Proxy Method Generation Process

execution units in order to better take advantage of the underlying hardware for improved performance and greater flexibility in realizing the identified concurrency in the FOP design.

As seen in Figure 9, a single mixin layers application can decomposed for concurrent execution in many ways. At a basic level, both layers (Figure 9(a)) and inner classes (Figure 9(b)) offer simple and orthogonal delineations of concurrent execution. Combining these orthogonal units of encapsulation geometrically increases the level of concurrency (Figure 9(c)).

This is not to say though that concurrent units can only be defined at the intersection of a layer and an inner class. On the contrary, because `Concurrent` is just another layer, the programmer can choose to logically combine any "neighboring" layers into a single concurrent unit, as shown in Figure 9(d). This flexibility improves performance in two ways: 1) by allowing tightly coupled objects to be conglomerated in order to facilitate inter-procedural optimizations (e.g., function inlining and register assignment); 2) by enabling concurrency throttling, an optimization where *fewer than* M processes performs better than M processes (13).

4.3 Introducing Concurrency Orthogonally

Concurrent mixin layers introduces concurrency orthogonally to the high-level design by using the active object pattern (26) for each concurrent execution unit and through the use of futures at the method level (21). That is, multiple methods, with at most one per concurrent unit, execute simultaneously returning a reference to a value that will *eventually* be computed. These technologies facilitate the extension and maintenance of the application orthogonally to the mechanisms providing low-level concurrency.

To create and synchronize concurrent tasks, we implement a "wait-by-necessity" policy using single-assignment futures. A future blocks if the client code attempts to retrieve its value before the value is defined. Using futures as the return type of every proxy method allows methods to execute concurrently by avoiding the immediate data-dependency. Implementing each future as a single-assignment datatype also improves performance and concurrency, as multiple active objects can simultaneously access the same future once the value is defined.

Moreover, our implementation transfers "execution ownership" between different concurrent execution units using "functoids" (4). This is required due to the caller and the callee executing within different execution contexts when a method has been intercepted by `Concurrent`. To provide the callee with the requisite parameters, a functoid encapsulates the caller's input parameters and future return value. The caller then pushes the functoid onto the callee's execution queue and continues executing (concurrently with the callee) until it invokes `value()` on any future.

4.3.1 Preserving Application Semantics

While futures and active objects facilitate concurrent execution, a simple implementation of these approaches is insufficient for our purposes. We wish to guarantee that the original semantics of the application are preserved, while simultaneously hiding low-

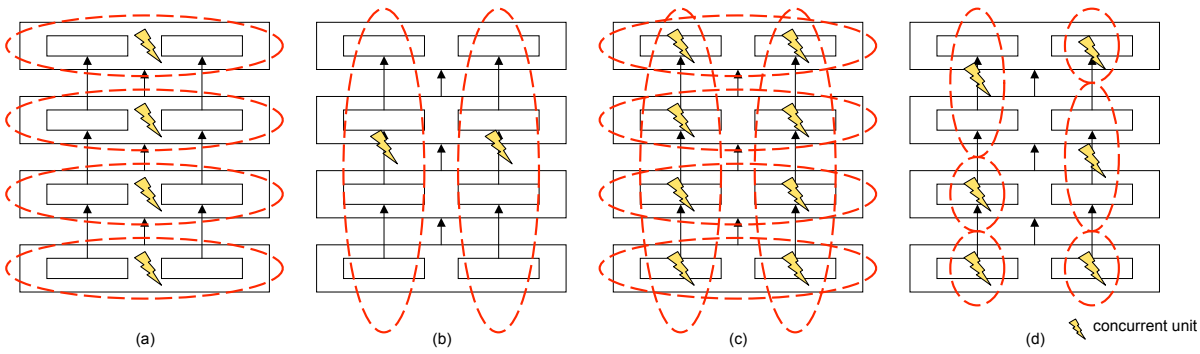


Figure 9. Decomposing Concurrent Mixin Layers to Achieve Multiple Granularities of Parallelism

level concurrency details (necessary in stateful languages) from the programmer's view. For example, consider the methods `add` and `sub`, as shown in Figure 10.

```
struct SodaDispenser {
    int numSodas, addCalled, subCalled;

    SodaDispenser() : numSodas(0),
                addCalled(0), subCalled(0) { }

    Future<bool> add() {
      addCalled++;
      numSodas ++;
      return Future<true>
    }

    Future<bool> sub() {
      subCalled++;
      if (numSodas > 0) numSodas--;
      return Future<true>
    }
};

int main{int argc, char** argv} {
  SodaDispenser machine;
  Future<void> f = machine.add();
  Future<void> g = machine.sub();
  f.value(); g.value();
  return machine.numSodas;
}
```

Figure 10. Potentially Ambiguous Execution Behavior

Since both `add` and `sub` modify the state variable `numSodas`, the value of `numSodas` is ambiguous depending on the interleaving of the accesses to `numSodas`. When, `add` completely precedes `sub`, `numSodas` equals 0; when, `sub` completely precedes `add`, `numSodas` equals 1. When the instructions for `add` and `sub` are interleaved in some fashion, `numSodas` depends on the exact interleaving. This ambiguity can be resolved by introducing a low-level synchronization mechanism around the access of `numSodas`. However, in the absence of a parallelizing compiler, this approach requires that the programmer explicitly introduce the required synchronization, a burden concurrent mixin layers aims to remove.

To alleviate this burden on the programmer and still fulfill the original application semantics without deadlock, concurrent mixin layers refines the active object pattern in the following ways:

1. Each active object completely fulfills each future in the order that it is placed in its first-in-first-out (FIFO) execution queue.
2. A future can only reach the necessary active object by traversing the feature stack in a stepwise fashion.
3. Argument parameters are evaluated eagerly.

While the above invariants can affect the level of concurrency available, we will demonstrate in Section 5 that reasonable performance can still be achieved.

We now present, through proof-by-contradiction, how total ordering is preserved and how deadlocks are avoided.

Preserving Total Ordering Suppose we have the feature stack, `A < Concurrent < B < Concurrent < C > > >`. Let, `A.main()` issue `B.f()` followed by `B.g()` followed by `C.h()`.

Proposition: Total ordering is always preserved.

Proof (by contradiction): Assume to the contrary that there exists an alternative ordering whereby `C.h()` will be executed prior to either `B.f()` or `B.g()`. By invariant 2, there is only one path for `C.h()` to reach `AO_C`, which is traversing from `AO_A` to `AO_B` and then `AO_B` to `AO_C`. Furthermore, by invariant 1, `C.h()` will only traverse to `AO_C` when reaching the front of `AO_B`'s FIFO queue. For `C.h()` to execute prior to either `B.f()` or `B.g()`, one of these functions must still be on `AO_B`'s FIFO queue. This implies that this function traversed from `AO_A` to `AO_B` after `C.h()` did and similarly this function must have been behind `C.h()` on `AO_A`'s queue according to invariant 1. For this to occur, `C.h()` must have been issued before this function, however, `C.h()` was issued after *both* `B.f()` and `B.g()` creating a contradiction.

Avoiding Deadlock Suppose we have the the same feature stack, as above.

Proposition: Deadlock is always avoided.

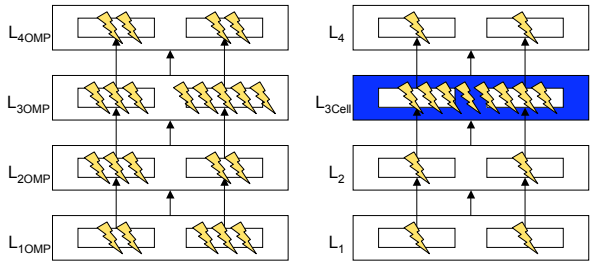
Proof (by contradiction): Assume that deadlock occurs such that `B.f()` depends on the value of `C.g()` yet `C.g()` is in the FIFO queue *after* `B.f()`. For `B.f()` to depend on the value of `C.g()`, `B.f(C.g())` must have occurred. Invariant 3 stipulates that parameters are eagerly evaluated, and thus `C.g()` must be issued and placed in the FIFO queue *before* `B.f()`. This contradicts our assumption that `C.g()` is in the FIFO queue *after* `B.f()`.

4.4 Enabling Optimizations for Expert Programmers

Although concurrent mixin layers strives to enable the general programmer to obtain reasonable performance with minimal knowledge of low-level concurrency details, it also enables expert programmers to optimize the performance even further. For example, Figure 9 shows how concurrency is geometrically proportional to the number of layers (P) and inner classes (Q). However, consider the case in which the underlying hardware contains M cores, where $M >> P * Q$.

Recall from Section 4.2 that concurrent mixin layers only *logically* decomposes the application into concurrent execution units

but does not specify *how* to implement each unit. Therefore, an expert programmer can create an explosion of parallelism by introducing parallelism *within* each layer using any complementary parallel technology such as OpenMP (15) (Figure 11(a)).



(a) Adding Parallelism within Layers with OpenMP

(b) Swapping in an IBM Cell Acceleration Layer

Figure 11. Two Approaches of Adding Parallelism within a Layer

Additionally, should N of the M cores reside within a specialized accelerator, such as the IBM Cell processor, a specialized accelerator module may be swapped in since concurrent mixin layers retains the modularity and extensibility benefits of mixin layers (Figure 11(b)).

5. Case Study

In some ways, concurrent mixin layers is an evolutionary technology. Thus, we present a case study that not only validates its applicability to the problem of developing parallel programs for multi-core systems, but also highlights the evolutionary path from PLAs and FOP to concurrent mixin layers. Specifically, we evaluate how our implementation enables a product line of multimedia applications to execute on an Intel quad-core system without requiring the programmer to manage low-level concurrency explicitly. We will also demonstrate that this approach effectively transforms a product line into parallel applications capable of utilizing multiple cores.

Our product line of multimedia applications is comprised of six features: `schedule`, `read`, `flip`, `invert`, `negative`, and `write`. Following FOP methodology, we map each feature to an individual layer, creating a feature-stack of up to height six. Despite the well-known advantages offered by a layered architecture, the resulting application is bound to using exactly one core if no parallelism has been added explicitly. In order to augment this product line for effective use on multi-core systems, we evolve the original implementation to concurrent mixin layers. A graphical view of the evolution of this multimedia product line to concurrent mixin layers is visible in Figure 12.

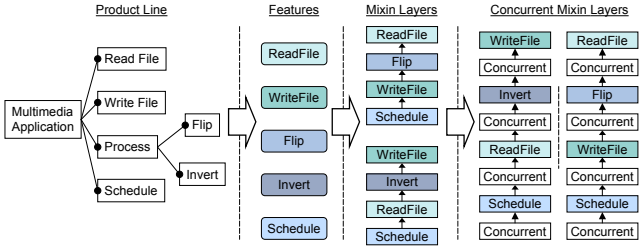


Figure 12. Evolution of a Multimedia Product Line to Concurrent Mixin Layers

In the following subsections, we document our experiences using concurrent mixin layers to execute a multimedia product line on a multi-core system.

5.1 Running N Features on N Cores

For the initial experimental setup, we chose to evaluate the performance of concurrent mixin layers using an application comprised of four features running on four cores. This setup provides a performance baseline by providing a perfect balance between features and cores. In addition, this setup makes it possible to determine how the ordering of layers can affect overall performance by varying the execution flow. A summary of our results is shown in Figure 13.

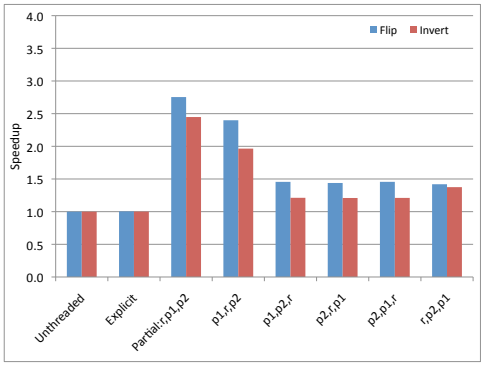


Figure 13. Speedup Achieved by Various Product-Line Applications on a Quad-core System

5.1.1 Explicit: Minimal Changes to the Original Code

To determine our performance baseline, we transformed the original implementation in a minimal way – changing the return type of every method into a future, and explicitly calling `value()` immediately after the method returns (Figure 14). The resulting code executes in lock-step similar to the original serial application.

```
for each(image* im in input) {
  image* r1 = read(im).value();
  image* p1= process(r1).value();
  image* w1 = process(p1).value();
  write(w1);
}
```

Figure 14. Example Code of "Explicit"

As one can see in Figure 13, the performance of this "explicit" concurrent mixin layers version exhibits an insignificant amount of computational overhead as compared to the original version. From this result, one can infer that the overhead incurred by concurrent mixin layers is minimal.

5.1.2 Partial: Using Futures to Overlap Computations

With our baseline established, we modify the product line to take advantage of futures, so that multiple methods may execute concurrently. As seen in Figure 15, `value()` is not invoked immediately following a method invocation. Instead, the methods are modified to accept futures as arguments.

```
for each(image* im in input) {
  Future<image*> f1 = read(im);
  Future<image*> f2 = process(f1);
  Future<image*> f3 = process(f2);
  write(f3);
}
```

Figure 15. Example code of "Partial"

While the introduced changes can increase the code's complexity (due to the use of futures), they do not change the overall algorithm's structure. As such, this "partial" parallelization achieves an average speedup of 2.3x on four cores (Figure 13). Although the speedup does not reach the ideal 4.0x, the ability to use more than one core improves the performance substantially as compared to the performance of a single-threaded version (i.e., 1.0x).

5.1.3 Pipelined: Leveraging the Layered Architecture

A performance analysis of the previous product lines shows that the ordering of the layers within the feature stack significantly affects the resulting speedup (Figure 13). When the layers are stacked bottom-to-top, in the same order that their features are utilized by the algorithm, a larger speedup can be realized. Leveraging this insight, one can identify a natural pipeline flow of execution inherent to the architecture.

Building on this observation, we refactor the algorithm of our multimedia application into dataflow pipelines (Figure 16). This approach exploits parallelism by treating each layer as a worker on a virtual assembly line, in which each worker operates on a different set of data simultaneously. A piece of data is considered fully processed after it has passed through the entire assembly line. A beneficial side affect of using a dataflow approach is that the resulting code tends to be shorter in length and less coupled.

```
//scheduler
for each(image* im in input){
  Future<image*> f1 = process(im);
}
...
Future<image*> process(image) {
  Future<image*> f1 = do_something(image);
  return Super::process(f1);
}
```

Figure 16. Example Code of "Pipelined"

Correspondingly, as the results show in Figure 17, this "pipelined" approach further improves performance. As such, we believe that using dataflow pipelines within concurrent mixin layers to be a promising approach for achieving high performance on multi-core systems.

5.2 Running N Features on M Cores

We now examine the situations in which the number of features does not equal the number of cores, to identify the adaptability of concurrent mixin layers under more typical conditions.

5.2.1 More Features Than Cores

In the case of a product line having more features than available cores, concurrent mixin layers allows the programmer to combine multiple layers into a single execution unit to provide better load balancing. For our tests, we evaluated the "invert" application, which consists of four total features, running on three cores and two cores, using three and two execution contexts, respectively.

These tests showed that performance varies significantly depending on which layers are combined. On three cores, the fastest performing combination was nearly 3x faster than the slowest combination; on two cores, the best combination was approximately 1.5x faster. Although, it is currently up to the programmer to decide how layers should be combined, a process that can be automated in the future, concurrent mixin layers greatly facilitates this process as the programmer simply has to change the `typedef` as shown in Figure 18.

```
typedef Concurrent C;
// Execution units: read/flip; invert; write
typedef C < Read < Flip < C < Invert < C
          < Write > > > > > > Object;
// Execution Units: read; flip; invert/write
typedef C < Read < C < Flip < C < Invert
          < Write > > > > > > Object;
```

Figure 18. Two Decompositions of the Same Feature Stack

5.2.2 Fewer Features Than Cores

As current hardware trends continue, it is possible that programmers will soon have many more cores than available features. As discussed earlier in Section 4.4, concurrent mixin layers allows the ability to add parallelism orthogonally to the number of features. However, at this time, we are unable to obtain computation time on an adequate many-core system and cannot evaluate concurrent mixin layers within this context.

5.3 Summary

From this case study, one can conclude that concurrent mixin layers delivers tangible performance and usability benefits to the programmer charged with the challenge of utilizing multi-core systems. The results show that it is possible to achieve improved performance with varying levels of modification to the original code without requiring the programmer to add low-level concurrency mechanisms explicitly. It has also been made apparent that one can achieve improved performance benefits when one is willing to transform their imperative algorithms to dataflow pipelines.

6. Related Work

While parallel programming has only recently entered mainstream computing with multi-core technology, it has been the prerogative of high-end computing (HEC) for several decades. In particular, HEC scientists have explored a plethora of parallel programming technologies including vector machines, shared memory multiprocessors, distributed memory computing cluster, grids of computing clusters, massively parallel supercomputers, and hybridization of the above (28).

Today parallel programming is dominated by two primary models of parallel computing: threading and message passing. The threading model relies on each concurrent unit having access to the entire global memory space, with OpenMP (15) and PThreads (19) being the predominant implementations. While our current implementation, concurrent mixin layers, utilizes PThreads as its concurrency mechanism, the higher level of abstraction of FOP leaves the programmer unaware of this low-level implementation detail.

For message passing, the message passing interface (MPI) (18), is becoming the de facto standard for parallel programming due to its support for both shared and distributed memory computers with a "write once, run anywhere" advantage. However, while MPI programs can achieve high performance, its programming abstractions are often found to be too low-level for experts, let alone the general programmer (22). Similarly to the threading model mentioned above, concurrent mixin layers offer a higher level of abstraction to the programmer; as an alternative low-level concurrency implementation, we plan to evaluate the use of MPI, as discussed in Section 7.

Recent trends in the HEC community have shifted the emphasis from strictly high-performance to high-productivity, in which performance and ease of development and maintenance play equally important roles (25). To realize this vision, several high-performance, high-*productivity* languages and technologies are being explored (2; 11; 12; 16; 30; 38). These new program-

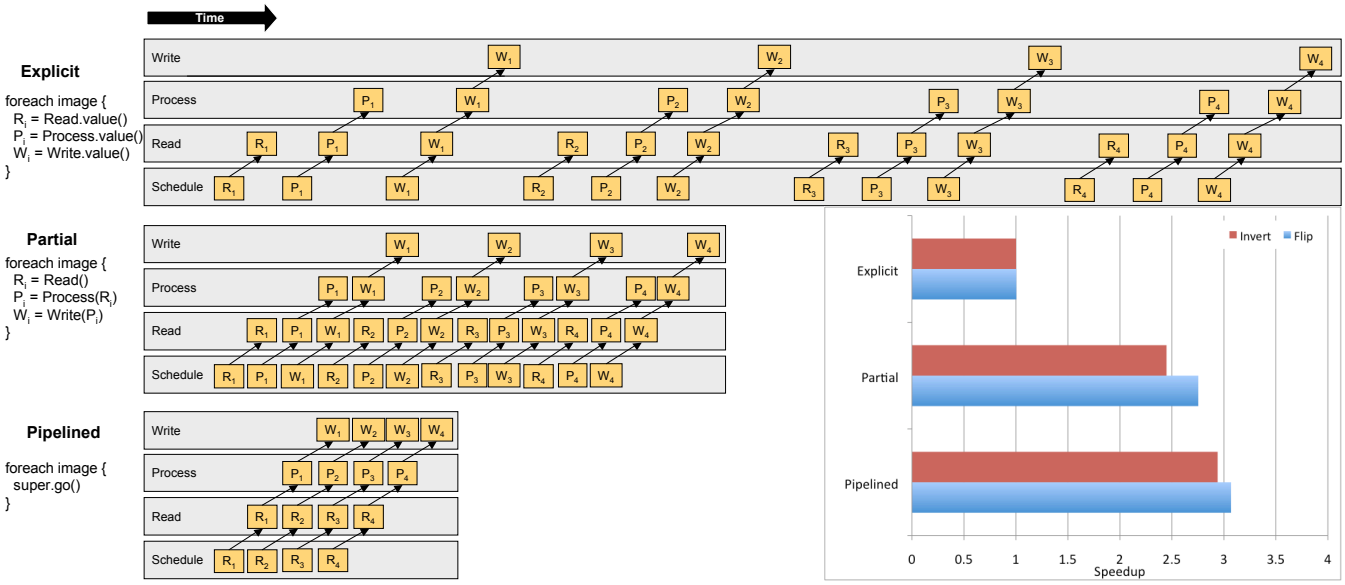


Figure 17. Leveraging the Layered Architecture to Minimize Pipeline Stalls

ming languages and technologies offer new specialized constructs and abstractions, designed specifically for high-productivity parallel programming. By contrast, concurrent mixin layers offer similar productivity advantages, but remains within the confines of existing mainstream programming languages. We believe that our approach aligns well with the recent insights gained by software engineering researchers, who point out that the average developer is resistant to changing their development tools and languages (7). Fully accommodating this resistance to change in the face of the radical change to multi-core processing may not be an ideal compromise, but in our view, allowing incremental changes toward the next-generation parallel programming technologies is a pragmatic solution.

Concurrent mixin layers also bears similarity to a hybridization of the Common Component Architecture (CCA) (6) and Sequoia (17). Like CCA, concurrent mixin layers modularizes functionality (e.g., features) and can expose a dataflow model to the programmer. Unlike CCA, however, concurrent mixin layers does not impose a strict API on the programmer. Instead, the programmer is free to choose an interface that is most beneficial for the task at hand. In addition, concurrent mixin layers enables the programmer to combine multiple units of functionality into a single "meta-component" at will.

As for Sequoia, the hierarchical abstract machine model resembles a concurrent mixin layers feature stack, in which communication occurs only in a vertical fashion. Nevertheless, Sequoia provides parallel abstractions for the hierarchical memory model, whereas concurrent mixin layers aims to provide abstractions for multi-core processors.

Lastly, concurrent mixin layers closely resembles an actor model (1). However, the invariants listed in Section 4.3.1 would restrict an actor model so significantly, that the resulting concurrent execution unit would no longer provide the capabilities one would expect from an actor, such as dynamic out-of-order execution.

7. Future Work

As future work, we plan to improve the generality, scalability, and usability of our methodology and implementation.

While the current implementation of concurrent mixin layers utilizes PThreads as the vehicle for concurrent execution, we are interested in exploring whether alternative low-level parallel technologies such as MPI (18) could be used instead and their affect on performance. We also plan to evaluate how languages other than C++ can serve as an implementation platform. Specifically, implementations in the mainstream language such as Java, Lisp, and Python as well as emerging languages such as X10 (12) or Scala (31) could provide valuable insights.

Our performance evaluation assessed the advantages of concurrent mixin layers on a "small" multi-core system. An evaluation on larger many-core systems such as the 8-core (32-thread) Sun Niagara2, the 64-core Tilera TILE64, and the 80-core Intel Teraflops Research Chip, will provide insights into the applicability of this approach on massively parallel hardware. Simultaneously, we will evaluate larger product lines to leverage the larger number of available cores.

Lastly, we plan to experiment with state-of-the-art generative programming technologies in aiding the transformation of product lines into parallel programs. In particular, the technologies of FeatureC++ (4) and Morphing (23) look especially promising.

8. Conclusions

We presented concurrent mixin layers, a novel methodology for constructing parallel applications that effectively utilizes the available processing power on multi-core systems. By using feature-oriented programming, a general programmer can readily identify concurrency within a product-line architecture, and through higher-level parallel programming abstractions can transform a serial mixin layers implementation into a parallel program. Furthermore, the transformation allows for multiple granularities of parallelism, but does not require the programmer to introduce low-level concurrency constructs explicitly. To demonstrate that our implementation can improve performance by leveraging multiple cores in parallel, we applied concurrent mixin layers to a product line of multimedia applications and realized a speedup of up to slightly over 3x on four cores. Unsurprisingly, since the resulting architecture closely resembles a processing pipeline, a dataflow programming model within this architecture achieves the best speedup. As such, we believe concurrent mixin layers can be an enabling plat-

form for efficient and intuitive parallel programming on multi-core systems.

References

[1] AGHA, G. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[2] ALLEN, E., CHASE, D., HALLETT, J., LUCHANGO, V., MASSEN, J.-W., RYU, S., STEELE, G., AND TOBIN-HOCHSTADT, S. The Fortress language specifications, v1.0, 2008.

[3] AMDAHL, G. M. The validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings* (1967), pp. 483–485.

[4] APEL, S., LEICH, T., ROSENMÜLLER, M., AND SAAKE, G. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *GPCE* (2005), pp. 125–140.

[5] APEL, S., LEICH, T., AND SAAKE, G. Aspectual mixin layers: Aspects and features in concert. In *International Conference on Software Engineering (ICSE)* (2006).

[6] ARMSTRONG, R., GANNON, D., GEIST, A., KEAHEY, K., KOHN, S., MCINNES, L., PARKER, S., AND SMOLINSKI, B. Toward a common component architecture for high-performance scientific computing. In *HPDC '99: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing* (Washington, DC, USA, 1999), IEEE Computer Society, p. 13.

[7] BASILI, V. R., CARVER, J., CRUZES, D., HOCHSTEIN, L., HOLLINGSWORTH, J. K., SHULL, F., AND ZELKOWITZ, M. V. Understanding the high performance computing community: A software engineers perspective. *IEEE Software 25*, 4 (2008).

[8] BATORY, D., JOHNSON, C., MACDONALD, B., AND VON HEEDER, D. Achieving extensibility through product-lines and domain-specific languages: a case study. *ACM Transactions on Software Engineering and Methodology 11*, 2 (2002), 191–214.

[9] BOSCH, J. *Design and use of software architectures: adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[10] BRACHA, G., AND COOK, W. Mixin-based inheritance. In *ECOOP/OOPSLA* (1990), pp. 303–311.

[11] CHAMBERLAIN, B., CALLAHAN, D., AND ZIMA, H. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications 21*, 3 (2007), 291–312.

[12] CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications* (New York, NY, USA, 2005), ACM, pp. 519–538.

[13] CURTIS-MAURY, M., WANG, T., ANTONOPOULOS, C., AND NIKOLOPOULOS, D. Integrating Multiple Forms of Multithreaded Execution on SMT Processors: A Quantitative Study with Scientific Workloads. In *Proc. of the Second International Conference on the Quantitative Evaluation of Systems (QEST'2005)* (Torino, Italy, Sept. 2005), pp. 199–208.

[14] CZARNECKI, K., AND EISENECKER, U. W. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[15] DAGUM, L., AND MENON, R. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering 05*, 1 (1998), 46–55.

[16] EL-GHAZAWI, T., CARLSON, W., AND DRAPER, J. UPC language specifications, v1.1.1, 2003.

[17] FATAHALIAN, K., KNIGHT, T. J., HOUSTON, M., EREZ, M., HORN, D. R., LEEM, L., PARK, J. Y., REN, M., AIKEN, A., DALLY, W. J., AND HANRAHAN, P. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (2006).

[18] FORUM, M. P. I. MPI: A message-passing interface standard, 1994.

[19] GALLMEISTER, B. O., AND LANIER, C. Early experience with POSIX 1003.4 and POSIX 1003.4 A. *Proceedings of the 12th Real-Time Systems Symposium* (1991), 190–198 (of ix + 307).

[20] GCC-XML, the XML output extension to GCC, 2007. http://www.gccxml.org/.

[21] HENRY C. BAKER, J., AND HEWITT, C. The incremental garbage collection of processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages* (New York, NY, USA, 1977), ACM, pp. 55–59.

[22] HOCHSTEIN, L., CARVER, J., SHULL, F., ASGARI, S., BASILI, V., HOLLINGSWORTH, J., AND ZELKOWITZ, M. HPC programmer productivity: A case study of novice HPC programmers. In *ACM/IEEE Supercomputing Conference (SC'05)* (2005).

[23] HUANG, S. S., ZOOK, D., AND SMARAGDAKIS, Y. Morphing: Safely shaping a class in the image of others. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (Aug. 2007), Springer-Verlag, pp. 399–424.

[24] KÄSTNER, C., KUHLEMANN, M., AND BATORY, D. Automating feature-oriented refactoring of legacy applications. In *Poster presented at Europ. Conf. Object-Oriented Programming (ECOOP)* (July 2007).

[25] KEPNER, J. HPC productivity: An overarching view. *International Journal of High Performance Computing Applications 18*, 4 (2004), 393–397.

[26] LAVENDER, R. G., AND SCHMIDT, D. C. Active object: an object behavioral pattern for concurrent programming. *Proc.Pattern Languages of Programs,* (1995).

[27] LIU, J., BATORY, D., AND LENGAUER, C. Feature oriented refactoring of legacy applications. In *ICSE '06: Proceedings of the 28th international conference on Software engineering* (New York, NY, USA, 2006), ACM, pp. 112–121.

[28] Tutorial: Introduction to parallel computing, 2007. https://computing.llnl.gov/tutorials/parallel_comp/.

[29] MANFERDELLI, J. L. The many-core inflection point for mass market computer systems. *CTWatch Quarterly* (2007).

[30] NUMRICH, R. W., AND REID, J. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum 17*, 2 (1998), 1–31.

[31] ODERSKY, M., ALTHERR, P., CREMET, V., EMIR, B., MANETH, S., MICHELOUD, S., MIHAYLOV, N., SCHINZ, M., STENMAN, E., AND ZENGER, M. An overview of the Scala programming language. *LAMP-EPFL* (2004).

[32] PARNAS, D. L. On the design and development of program families. *IEEE Transactions on Software Engineering 2*, 1 (1976), 1–9.

[33] PREHOFER, C. Feature-oriented programming: A fresh look at objects. In *ECOOP* (1997), pp. 419–443.

[34] SMARAGDAKIS, Y., AND BATORY, D. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodologies (TOSEM) 11*, 2 (2002), 215–255.

[35] SPINCZYK, O., AND LOHMANN, D. The design and implementation of AspectC++. *Knowledge-Based Systems 20*, 7 (2007), 636–651.

[36] THIEL, S., AND HEIN, A. Modeling and using product line variability in automotive systems. *IEEE Software 19*, 4 (2002), 66–72.

[37] WEISS, D. M., AND LAI, C. T. R. *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[38] YELICK, K., SEMENZATO, L., PIKE, G., MIYAMOTO, C., LIBLIT, B., KRISHNAMURTHY, A., HILFINGER, P., GRAHAM, S., GAY, D., COLELLA, P., AND AIKEN, A. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing* (New York, NY 10036, USA, 1998), ACM, Ed., ACM Press.