Building Scalable Software Systems in the Multicore Era

Hridesh Rajan
Iowa State University,
226 Atanasoff Hall,
Ames, IA, USA
hridesh@iastate.edu

ABSTRACT

Software systems must face two challenges today: growing complexity and increasing parallelism in the underlying computational models. The problem of increased complexity is often solved by dividing systems into modules in a way that permits analysis of these modules in isolation. The problem of lack of concurrency is often tackled by dividing system execution into tasks that permits execution of these tasks in isolation. The key challenge in software design is to manage the explicit and implicit dependence between modules that decreases modularity. The key challenge for concurrency is to manage the explicit and implicit dependence between tasks that decreases parallelism. Even though these challenges appear to be strikingly similar, current software design practices and languages do not take advantage of this similarity. The net effect is that the modularity and concurrency goals are often tackled mutually exclusively. Making progress towards one goal does not naturally contribute towards the other. My position is that for programmers that are not formally and rigorously trained in the concurrency discipline the safest and most productive way to get scalability in their software is by improving modularity of their software using programming language features and design practices that reconcile modularity and concurrency goals. I briefly discuss preliminary efforts of my group, but we have only touched the tip of the iceberg.

1. PROBLEMS AND THEIR IMPORTANCE

Scalability of software in the next decade crucially depends on its ability to effectively utilize multicore platforms [5]. For scientific applications such scalability generally comes from invention and refinement of better algorithms and data structures, but that is not the case for non-scientific software that often exhibit irregular fine-grained parallelism. However, scalability of these applications is an equally important concern for society, defense, and the individual.

Scalability of these applications faces two major hurdles. A first and well-known hurdle is that writing correct and efficient concurrent software using *concurrency-unsafe* programming language features has remained a challenge [10]. A language feature is concurrency-unsafe if its usage may give rise to program execution sequences that contains two or more memory accesses to the same location that are not ordered by a happens-before relation and at least one is a write to the memory [7]. Threads and processes are examples of such features as are `Future` and `FutureTask` as embodied in the 1.5 edition of the Java programming language [11, 21]. Without strict design and implementation disciplines they are concurrency-unsafe. Many of the features planned for 1.7 edition of the Java programming language are similarly concurrency-unsafe.

A second and less explored hurdle is that unlike in scientific applications, in general-purpose programs potential concurrency isn't always obvious. A typical scientific application is generally data-parallel, whereas general-purpose programs typically exhibit irregular parallelism. As a result, techniques that have been remarkably successful in scientific domains have only seen modest success for general-purpose programs [6].

I believe that both these hurdles persist, in part, because of a significant shortcoming of current software design practices. The basic problem is that modularity and concurrency are treated as two separate and orthogonal goals. As a result, concurrency goals are often tackled at a level of abstraction lower than modularity goals. Synchronization defects arise when developers work at low abstraction levels and are not aware of the behavior at a higher level of abstraction. This lack of awareness also limits the discovery of potentially available concurrency in the resulting systems.

All of this is complicated by the fact that our current software development workforce is vastly under-prepared to develop correct, efficient and fair software systems for the emerging multicore hardware platforms using concurrency-unsafe features that are currently available in languages and libraries.

In the rest of this paper I explain my position. Section 2 briefly describes key insights and discusses preliminary efforts of my research group towards enabling scalable software and scalable software engineering for emerging multicore platforms. In Section 3 I discuss implications of our observations and Section 4 concludes.

2. HOW TO SOLVE IT?

Look at the unknown! And try to think of a familiar problem having the same or a similar unknown. — George Pólya, 1945.

Are better software designs inherently more concurrent? In the following I will argue and present some evidence to my hypothesis that modularity and concurrency goals are intertwined and that by advances in programming language design and software design practices, it may be possible to achieve mutualism between them!

To motivate consider a simple example shown in Figure 1, which shows three versions of the parts of a telecommunication software. The class `Call` shown in this figure models a typical connection in such setting. It models the state of a phone call using enumeration `State` and the caller and the receiver with fields `caller`

To appear as: Hridesh Rajan, "Building Scalable Software Systems in the Multicore Era", In the proceedings of the 2010 FSE/SDP Workshop on Future of Software Engineering, Santa Fe, New Mexico, November 2010.

Non-Modular Version

```
class Call {
 enum State { PENDING,
       COMPLETE, DROPPED }
 Customer caller, receiver;
 State state = PENDING;
 Call(Customer a, Customer b) {
    caller = a; receiver = b;
  }
 void complete() {
   state = COMPLETE;
   timer.start();
  }
 void drop() {
  state = DROPPED;
  timer.stop();
  long time = timer.getTime();
  long cost = 0.07 * time;
  caller.addCharge(cost);
  }
 Timer timer = new Timer();
}
```

Modular Version

```
interface CallEndObserver {
  void notify (Customer c, Timer t);
}
class Call {
 /*Omitted fields and methods
     same as the code on left.*/
 void drop() {
  state = DROPPED;
  timer.stop();
  for(CallEndObserver o : observers)
   o.notify(caller,timer);
 }
 List<CallEndObserver> observers;
 void addObserver(CallEndObserver o){
  observers.add(o);
 }
 void removeObserver(CallEndObserver o){
  observers.remove(o);
 }
}
class Billing implements CallEndObserver {
 void notify (Customer c, Timer t) {
  long time = timer.getTime();
  long cost = 0.07 * time;
  caller.addCharge(cost);
 }
}
```

Modular + Concurrent Version in Pāñini [8].

```
event CallEnd {
  Customer c; Timer t;
}
class Call {
 /*Omitted fields and methods
     same as the code on left.*/
 void drop() {
  state = DROPPED;
  timer.stop();
  announce CallEnd(caller,timer);
 }
}
class Billing {
 when CallEnd do notify;
 void update (Customer c, Timer t) {
  long time = timer.getTime();
  long cost = 0.07 * time;
  caller.addCharge(cost);
 }
}
```

Figure 1: Modularization of the billing requirement (left → middle) makes concurrent solution (right) evident.

and `receiver` respectively. It also contains a `timer` object to monitor the duration of a call. This class provides two methods `complete` and `drop` that serve to connect and disconnect a call respectively.

An example requirement for such application would be to bill customers for the duration of the conversation. A simple implementation of such requirement could be done by adding its logic to the code for `drop` method (shown in the left listing as the highlighted code).

This solution works, however, it has several software engineering problems. For example, since the code for billing is mixed with the code for call logic, it would be harder to implement any changes to either requirement. This is primarily because the developer making changes to either requirement would have to understand the other requirement as well to ensure correctness. In addition, implementation of neither requirements is reusable. Last but not least, understanding billing and call logic in isolation is not possible because their implementations are mixed.

This example demonstrates, at a small scale, the modularity problems faced by developers in building large software systems. To modularize the implementation of the billing requirements, a good software engineer would separate its implementation out in a new module, while ensuring that this new module communicates to the class `Call` via a well-defined interface (and vice-versa). The middle column shows the modularized version, where the implementation of the billing requirement is separated out as the class `Billing` using the Observer design pattern [3].

The solution in the middle is modular and solves all the problems pointed out previously with the solution in the left column. In this version, the code for billing and call logic are separated via well-defined interface in the form of event type `CallEnd`. This makes it easier to change these independently, reuse them, and understand them. This design thus breaks the dependencies between the implementation of these two requirements.

Quite interestingly, this design can facilitate the concurrent execution of the billing logic. For example, we could encapsulate the shaded area in the middle column to run as a concurrent task. In other words, the modularization transformation from the left to the middle could also serve as an effective parallelization transformation. The key research question is whether the observation made in the context of this example holds for a large class of requirements.

This question rests on the observation that from the point of view of both concurrency and modularity, challenges are similar. For example, in order for the modular reasoning about the class `Billing` to succeed, it is important to understand the explicit and implicit dependencies of the billing concern. Similarly, in order for concurrent processing of billing to succeed one must also understand these dependencies to avoid data races and deadlocks in the solution that can potentially decrease parallelism. It is thus intuitive that the lack of modularity in design has direct ramifications on the available concurrency. However, it is not clear at this moment, whether improved modularity in a software design helps with concurrency in general. We now briefly discuss our preliminary efforts to further understand this duality. A detailed description of these ideas appears in the following papers [8, 13].

2.1 Asynchronous, Typed Events

Along one direction, we have developed the notion of asynchronous, typed events [8] in our language Pāñini that reconciles the modularity goal promoted by the implicit invocation design style [4, 20] with concurrency goals. Pāñini's design is inspired from my previous work on Ptolemy [14] and Eos languages [12, 15–17].

In implicit invocation design style, some modules (called subjects or publishers) signal events, e.g. reaching a program point, a condition becoming true, etc. Other modules (called observers or subscribers) express interest in receiving notifications when an event is signaled. The key advantage is that subjects can notify such observers without knowing about them (implicitly). Thus, implicit invocation design style decouples subjects and observers.

Asynchronous, typed events provide implicit concurrency in program designs when events are signaled and consumed without the need for explicit locking of shared states. The semantics is similar to other proposals based on message-based communication between concurrent tasks such as in Erlang [1], however, unlike these actor-based/message-based languages, Pāñini does not require complete isolation of such tasks. Furthermore, the communication between implicitly concurrent tasks is not limited to value types or record of value types.

The implementation in Figure 1, right uses the features of the Pāñini language [8]. The method `drop` in this implementation announces an event of type `CallEnd`. The declaration of the type of this event is shown at the top of the middle column. The class `Billing` features a new construct in Pāñini called *binding* (**when** `CallEnd` **do** `...`). This construct says to run the method `update` whenever any event of type `CallEnd` is announced in any class (for instance such event is announced in the class `Call`). As a result, whenever a call ends, the billing information is computed and updated concurrently. Pāñini provides race and deadlock freedom and a sequential semantics [8].

We have implemented a compiler and runtime system for Pāñini that is available for general distribution from the URL: `http://paninij.org` . We have tried out several programs, where asynchronous, typed events improve both modularity in program design and potentially available concurrency. Our performance results show that the generated code for Pāñini programs perform as well as their hand-tuned concurrent implementation [8].

2.2 GOF Object-oriented Design Patterns

Along another direction, we are developing a concurrent design pattern framework [13] that is attempting to reconcile modularity and concurrency goals by enhancing Gang-of-Four (GOF) object-oriented design patterns [3]. GOF patterns are commonly used to improve the modularity of object-oriented software. These patterns describe strategies to decouple components in design space and specify how these components should interact.

We have enhanced these patterns to also decouple components in execution space, so applying them concomitantly improves the design and potentially available concurrency in software systems.

For 18 out of the 23 GOF patterns, we have determined that, subject to appropriate usage, our hypothesis is true. For each of these 18 patterns we have created an enhanced version of the pattern in which use of the pattern increases potential concurrency without additional, explicit effort on the part of the developer to do so. In every case but one, the concurrency-related concerns (such as thread creation and synchronization) are fully encapsulated in a library that we provide, and in no case is the developer ever required to explicitly create a thread or acquire a synchronization lock.

A preliminary release of our framework is available for general distribution from the URL: `http://paninij.org/patterns/`.

2.3 Summary of Preliminary Efforts

The preliminary efforts of my group towards the design of the Pāñini language and the Pāñini concurrent pattern framework shows the feasibility of my hypothesis that by advances in programming language design and software design practices, it may be possible to achieve mutualism between modularity and concurrency goals.

3. IMPLICATIONS

Encouraged by these preliminary results my students and I seek generalization of our observations: what properties does a modularization transformation need to have to also make it an effective parallelization transformation? To what extent can we adapt/use traditional modularization techniques from software engineering to achieve modularization and concurrency at the same time? If not, is there a mismatch between current modularization techniques and the concurrency models? What advances in modularization techniques and language designs are necessary to address this mismatch? How can one capitalize on design benefits to yield concurrency? Are there any helpful design disciplines?

3.1 Maintenance and Reuse of Software

Mainstream programmers have just started to develop software that aims to effectively utilize multicore and manycore CPUs. One of the challenges that we have yet to face is maintenance of such software. I fully expect concurrent software to suffer from version maintenance nightmare in a manner similar to those typically seen in unmanaged languages (C, C++, etc). This is because, computer architecture variations are abound. One vendor (Intel) alone has shipped around 10 different multicore processors between 2004 - 2010 with substantially different characteristics (e.g. L1 cache sizes ranging from 16KB - 12MB). Writing explicitly concurrent software for these platforms generally requires careful calibration. For example, to match the number of threads to available cores, to match the data locality to cache sizes, etc. Since there are often significant performance gains to be had, it is natural to start seeing different versions fine-tuned to specific multicore CPUs [18, 19].

Achieving synergy between modularity and concurrency goals can potentially help with this problem. This synergy exposes implicit potential concurrency in program design creating candidates with richer information that can potentially be analyzed by underlying runtime environment. Take our concurrent design pattern framework as an example. Each GOF design pattern implementation in this framework helps expose potential concurrency between pattern participants but doesn't dictate concrete mapping to threads/locks, etc. So given the potential concurrency in program and the actual concurrency provided by the platform, the runtime environment is free to choose most appropriate mapping between the two.

3.2 Testing, Formal Verification, and Analysis

Along another dimensions, synergy between modularity and concurrency in this manner can have significant implications on scalability of software verification processes. Verifying sequential programs is still difficult. Verifying concurrent programs can be a nightmare. Generally a programmer is concerned about three potential problems.

1. *Data races:* Is there an interleaving in this program that can lead to data races on certain variables?
2. *Deadlocks:* Can concurrent tasks in this program deadlock under certain circumstances?
3. *Non-deterministic Semantics:* Can this program behave differently under distinct interleavings of tasks?

An approach using implicit concurrency must ensure that programs do not have these problems. The fundamental challenge with static verification of these conditions is that existing algorithms are imprecise and don't scale [2]. The precision and scalability issues in these techniques arise due to the unmanageable scope of analysis in large programs. This is because the analysis must consider all possible interleavings in the program and either prove that they satisfy the desired properties or declare certain interleavings as unsafe. Dynamic verification approaches have also been proposed, e.g. FastTrack [2], but the value of their output depends on the quality of the test cases. They are also not sound.

A research question then is that if (a) implicit concurrency is introduced using well-defined language features and design patterns and (b) implemented using a well-specified library such as the one we are proposing to develop, can analysis tools exploit the knowledge of the design pattern and specification of the language feature to narrow the scope of the analysis? To illustrate consider the observer design pattern. Participants in this pattern are subjects and observers. Let us assume that the specification of the interaction patterns among subjects and observers in the concurrency-enhanced observer pattern states that after announcing an event,

the subject must block until all observers have finished their tasks. This specification immediately narrows down the scope of the static analysis to interleavings between observers for a given event. If these observers do not have data races, do not deadlock, and have deterministic semantics then that particular application of the concurrency enhanced observer pattern will also have these properties.

The fundamental challenge then is in specifying the patterns, libraries, and language features in a manner that allows the use of this specification to narrow the scope of program analysis for scalability and precision of verification techniques.

3.3 Software Engineering Education

Implication on software engineering education are also noteworthy. Methods to educate application programmers in concurrency disciplines has recently attracted significant attention (See: First Workshop on Curricula in Concurrency and Parallelism, co-located with OOPSLA 2009 and similar first workshop on multicore education co-located with ASPLOS 2009). There is a sense of urgency towards incorporating similar topics into undergraduate curriculum around the world. A recent survey [9] by the Working Group on Software Engineering for parallel Systems found that 46 universities worldwide offered courses that discusses aspects of parallel programming. US ranked second in number of lectures after German universities. Furthermore, most offered material targeted graduate students. Worldwide, it was found that 26% of lectures were related to undergraduate courses, while the ratio drops to 23% in US universities.

On the other hand, topics on modularity and techniques to create modular software designs are an integral part of the graduate and undergraduate computer science curriculum for the last several decades. The key question then is whether achieving synergy between modularity and concurrency goals helps capitalize on existing expertise in teaching modularity-related topics to train next generation of software engineers in development of correct and efficient software for the multicore era.

4. CONCLUSION

Introducing concurrency has become important for the scalability of today's software systems, however, writing correct, efficient, and fair concurrent programs remains hard. In this work, I have taken the position that building programming language features and design practices that reconcile concurrency and modularity has the potential to solve this problem. Our initial work on the Pāñini language [8] and the concurrent design pattern framework [13] has demonstrated the feasibility of basic ideas, however much work remains to be done. We hope that the discussion at the 2010 FSE/SDP workshop on the Future of Software Engineering Research will help shed further light on these problems that have become important for the scalability of software systems in the multicore era.

Acknowledgments

This work was supported in part by the NSF under grant CCF-08-46059. Discussions with Steven M. Kautz, Yuheng Long, Sean Mooney, Tyler N. Sondag, Robert E. Dyer, and Wayne Rowcliffe was instrumental in developing and evolving these ideas.

5. REFERENCES

[1] J. Armstrong, R. Williams, M. Virding, and C. Wikstroem. *Concurrent Programming in ERLANG*. Prentice-Hal, 1996.

[2] Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. In *PLDI*, pages 121–133, 2009.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[4] David Garlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *the 4th Symposium of VDM Europe*, pages 31–44, 1991.

[5] David Geer. For Programmers, Multicore Chips Mean Multiple Challenges. *Computer*, 40(9):17–19, 2007.

[6] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI*, pages 211–222, 2007.

[7] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.

[8] Yuheng Long, Sean L. Mooney, Tyler Sondag, and Hridesh Rajan. Implicit invocation meets safe, implicit concurrency. In *Ninth International Conference on Generative Programming and Component Engineering*, Oct 2010.

[9] D. Meder, V. Pankratius, and W. F. Tichy. Parallelism in curricula an international survey. Technical report, University of Karlsruhe, 2008.

[10] J. Ousterhout. Why threads are a bad idea (for most purposes). In *ATEC*, January 1996.

[11] Polyvios Pratikakis, Jaime Spacco, and Michael Hicks. Transparent Proxies for Java Futures. In *OOPSLA*, pages 206–223, 2004.

[12] Hridesh Rajan. Design patterns in Eos. In *PLoP*, Sep 2007.

[13] Hridesh Rajan, Steven M. Kautz, and Wayne Rowcliffe. Concurrency by modularity: Design patterns, a case in point. In *Onward! Conference*, October 2010.

[14] Hridesh Rajan and Gary T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP*, pages 155–179, 2008.

[15] Hridesh Rajan and Kevin Sullivan. Eos: instance-level aspects for integrated system design. In *the European software engineering conference and international symposium on Foundations of software engineering (ESEC/FSE)*, pages 297–306, 2003.

[16] Hridesh Rajan and Kevin J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *the international conference on Software engineering (ICSE)*, pages 59–68, 2005.

[17] Hridesh Rajan and Kevin J. Sullivan. Unifying aspect- and object-oriented design. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 19(1), August 2009.

[18] Tyler Sondag, Viswanath Krishnamurthy, and Hridesh Rajan. Predictive thread-to-core assignment on a heterogeneous multi-core processor. In *PLOS*, Oct 2007.

[19] Tyler Sondag and Hridesh Rajan. Phase-guided thread-to-core assignment for improved utilization of performance-asymmetric multi-core processors. In *IWMSE*, May 2009.

[20] Kevin J. Sullivan and David Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–68, July 1992.

[21] Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe Futures for Java. In *OOPSLA*, pages 439–453, 2005.