Three hours

Two academic papers are provided for use with Section B

UNIVERSITY OF MANCHESTER
SCHOOL OF COMPUTER SCIENCE

Designing for Parallelism and Future Multi Core Computing

Date: Friday 25th January 2013
Time: 09:45 - 12:45

Please answer either:

any TWO Questions from the FOUR Questions provided in Section A

or

the ONE Question provided in Section B

PLEASE USE A SEPARATE ANSWERBOOK FOR EACH SECTION

This is an OPEN book examination.

The use of electronic calculators is permitted provided they are not programmable and do not store text.
SECTION A

Designing for Parallelism

Question 1

Consider the following algorithm which implements a 9-point finite difference stencil for the calculation of the solution of a time-dependent problem (as may be found, for example, in a weather modelling application):

\[
U_{i,j}^{(k+1)} = f \left( U_{i-1,j-1}^{(k)}, U_{i-1,j}^{(k)}, U_{i-1,j+1}^{(k)}, U_{i,j-1}^{(k)}, U_{i,j}^{(k)}, U_{i,j+1}^{(k)}, U_{i+1,j-1}^{(k)}, U_{i+1,j}^{(k)}, U_{i+1,j+1}^{(k)} \right) + g_{i,j},
\]

where \( g_{i,j} = g(x_i, y_j) \) is a constant (forcing) term (you may assume period (cyclic) boundary conditions in both \( x \) and \( y \) dimensions).

a) Consider a 1-dimensional partition of the \( x \) dimension of the 2-dimensional solution space into \( P \) blocks (block partition of the \( i \) index), where \( N \) is an integer multiple of \( P \), so that each task comprises the computation of \( N \times N/P \) components of the solution \( U_{i,j} \). (You should assume that the calculation is “in-place”, with the newly calculated \( U_{i,j} \) values overwriting the old values.)

i) Given that \( t_c \) is the average computation time for \( U_{i,j} \), determine a model of the total computation time of each step of the algorithm and, for \( P \) processors, the corresponding computation time per processor.

(2 marks)

ii) Given that \( t_s \) and \( t_w \) are the start-up cost of a message and the cost per word of transmission (note that the unknowns \( U_{i,j} \) are represented as single precision real variables that occupy one word of memory), derive (and carefully justify) a model of the total communication time for each step of the algorithm and, for \( P \) processors, the corresponding communication time per processor.

(4 marks)

iii) Derive an expression for the relative efficiency of the algorithm.

(2 marks)
b) Consider a 2-dimensional (square) partition of the $x$ and $y$ dimensions of the 2-dimensional solution space into $P$ blocks ((square) block partition of the $i$ and $j$ indices) so that each task again comprises the computation of $N^2/P$ components of the solution $U_{i,j}$. (Note: you should consider only partitions where the square root of $P$ is an integer, and assume that $P \leq N^2/4$.)

i) Given that $t_c$ is the average computation time for $U_{i,j}$, determine a model of the total computation time of each step of the algorithm and, for $P$ processors, the corresponding computation time per processor.

(2 marks)

ii) Given that $t_s$ and $t_w$ are the start-up cost of a message and the cost per word of transmission, respectively, derive a model of the total communication time for each step of the algorithm and, for $P$ processors, the corresponding communication time per processor.

(4 marks)

iii) Derive an expression for the relative efficiency of the algorithm.

(2 marks)

c) For the case where $\sqrt{P}$ is an integer, compare the number of messages required and the overall communication costs of the two partition models from parts a) and b).

(4 marks)
**Question 2**

The following (reduced) algorithm attempts to solve the Critical Section problem.

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>wantp ← false, wantq ← false</td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td>forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>p1:</td>
<td>wantp ← true</td>
<td>q1:</td>
</tr>
<tr>
<td>p2:</td>
<td>await wantq = false</td>
<td>q2:</td>
</tr>
<tr>
<td>p3:</td>
<td>wantp ← false</td>
<td>q3:</td>
</tr>
</tbody>
</table>

a) Briefly explain the term *control point* (in relation to each statement in the above algorithm), and identify the control points associated with the critical sections and non-critical sections in processes P and Q.

(2 marks)

b) i) Define a representation for the state of a computation of the algorithm and produce the full state space diagram for the (reduced) algorithm.

(4 marks)

ii) Use the state space diagram to justify whether the algorithm provides mutual exclusion.

(2 marks)

iii) Use the state space diagram to justify whether the algorithm is free from deadlock.

(2 marks)

c) The non-critical sections of the algorithm could contain many statements and thus take a long time to execute. On a computer with multi-core processors, the non-critical sections could, therefore, execute *at the same time* if each section was run on a different core. Describe how this (“true”) concurrency would appear in the state space diagram; thus explain how “true” concurrency is represented in the interleaving model of concurrency. Show a relevant fragment of the state space diagram for non-critical sections consisting of two statements in each process.

(5 marks)

d) In around half a page to a page identify, and comment upon, the technical challenges for software and hardware implied in the following quote on exascale computing (taken from the International Exascale Software Project Roadmap):

“It is clear that [exascale platforms] will embody radical changes along a number of different dimensions as compared to the architectures of today’s systems, and that these changes will render obsolete the current software infrastructure for large-scale scientific applications.”

(5 marks)
### Question 3

a) It is possible for the following synchronous channel-based algorithm to deadlock. Explain how this deadlock occurs and relate it to the four necessary and sufficient conditions for deadlock to occur. (The example may be thought of as three people, Alice, Bob and Chris attempting to communicate with each other. Sending true on a channel (chX <= true) is equivalent to calling the person whose initial appears in the channel name. Receiving from a channel (chX => dummy) indicates accepting a call (the algorithm receives into a dummy argument because we are not concerned with any value communicated).

<table>
<thead>
<tr>
<th>Communication Example</th>
<th>channel of boolean chA, chB, chC</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>boolean dummy</td>
<td>boolean dummy</td>
</tr>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>a1: chB &lt;= true</td>
<td>b1: chC &lt;= true</td>
</tr>
<tr>
<td>a2: chC =&gt; dummy</td>
<td>b2: chA =&gt; dummy</td>
</tr>
</tbody>
</table>

(4 marks)

b) The following algorithm attempts to fix the above problem by allowing ‘timeouts’ on attempts to communicate. You may assume there is a fourth process that will always engage in a communication on the timeout channel of each process. Show that deadlock is still possible in this algorithm by describing how the deadlock can occur and giving an execution trace leading to the deadlock. (Note: the “either…or…” statement used in the algorithm implements a selection between the alternatives. When the statement is executed, if communication can take place on more than one of the alternative channels, one of them is selected non-deterministically. If communication can take place on only one alternative channel, it is selected. Otherwise, the process blocks until communication can take place on one of the alternative channels.)

<table>
<thead>
<tr>
<th>Communication Example</th>
<th>channel of boolean chA, chB, chC, chAtimo, chBtimo, chCtimo</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>boolean dummy</td>
<td>boolean dummy</td>
</tr>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>either</td>
<td>either</td>
</tr>
<tr>
<td>a1: chB &lt;= true</td>
<td>b1: chC &lt;= true</td>
</tr>
<tr>
<td>a2: chC =&gt; dummy</td>
<td>b2: chA =&gt; dummy</td>
</tr>
<tr>
<td>or</td>
<td>or</td>
</tr>
<tr>
<td>a3: chAtimo &lt;= true</td>
<td>b3: chBtimo &lt;= true</td>
</tr>
<tr>
<td>a4: chC =&gt; dummy</td>
<td>b4: chA =&gt; dummy</td>
</tr>
</tbody>
</table>
c) i) Develop a truly deadlock free solution to the problem described in part a) using appropriate timeout actions. Correct algorithmic syntax is not required but your description of the proposed solution should be clear and unambiguous in its intended behaviour. (Note: a solution without timeouts will be explored in part d.).

(3 marks)

ii) Explain how examining the structure of the state space diagram of your solution would formally establish its freedom from deadlock.

(1 mark)

iii) Informally justify why your revised solution is deadlock free; you may provide fragments of the state space diagram if you wish. (An informal description is sufficient because the full state space diagram will almost certainly be too complex to draw manually in an exam.)

(2 marks)

iv) Is there a fairness issue with your solution? That is, is there a sequence of state transitions that could be selected to be executed infinitely often by an unfair scheduler, thereby preventing the system from making progress (i.e. preventing some calls from being made)? If so, give a short example of a sequence of the actions involved, otherwise justify why your solution is fair.

(4 marks)

d) Develop a deadlock free solution for the problem as stated in part a) that does not use timeouts. State which of the four necessary and sufficient conditions your solution breaks. Draw the full state space diagram, and hence show the solution is deadlock free. (The state space diagram may be as small as three states.)

(3 marks)
Question 4

a) Consider a multi-server queueing system with a single queue and three service points where the arrival rate is 75 customers per hour (and arrivals are assumed to satisfy a Poisson distribution). The average service time is 2 minutes per customer, and service times are assumed to be negative-exponentially distributed.

i) Calculate the expected waiting time, $W_q$.

(4 marks)

ii) Due to external factors, the average arrival rate of customers is expected to decrease and one of the three service points will be removed when to do so will mean that the expected waiting time remains less than 10 minutes. What arrival rate (customers per hour) will trigger the removal of one of the service points?

(6 marks)

b) Consider a queueing system with one service point where the arrival rate is $\lambda$ customers per unit time (and arrivals are assumed to satisfy a Poisson distribution). The average service rate is $\mu$ customers per unit time, service times are assumed to be negative-exponentially distributed, and $\rho = \frac{\lambda}{\mu} < 1$.

i) In terms of $\rho$, show that the probability of having to wait for service is

$$\Pr\{\text{wait for service}\} = \rho,$$

and that the mean queue length is

$$L_q = \frac{\rho^2}{1 - \rho}.$$  

(3 marks)

ii) In order to reduce the mean queue length, the current service point will be replaced with a new service point that has three times the capacity (i.e. the new service point has average service rate of $3\mu$ customers per unit time, and service times remain negative-exponentially distributed). Show that the mean queue length becomes

$$L_{q_{\text{new}}} = \frac{\rho^2}{3(3 - \rho)}.$$  

(4 marks)

iii) Further show that the relative reduction in the mean queue length is

$$\frac{L_{q_{\text{new}}}}{L_q} = \frac{(1 - \rho)}{3(3 - \rho)},$$

and comment on this ratio.  

(3 marks)
SECTION B

Future Multi-Core Computing

Question 1

Provide an analysis of one of the following two papers:

A) A Type and Effect System for Deterministic Parallel Java. OOPSLA 2009
or
B) CRUISE: Cache Replacement and Utility-aware Scheduling. ASPLOS 2012

by answering the following questions:-

a) What is the problem being addressed? (9 marks)
b) What is the proposed solution? (10 marks)
c) What are the assumptions? (4 marks)
d) How is it evaluated? (9 marks)
e) What are the limitations? (5 marks)
f) Overall assessment of paper and possible improvements? (3 marks)

(Total 40)

END OF EXAMINATION
A Type and Effect System for Deterministic Parallel Java

Robert L. Bocchino Jr. Vikram S. Adve Danny Dig
Sarita V. Adve Stephen Heumann Rakesh Komuravelli Jeffrey Overbey
Patrick Simmons Hyojin Sung Mohsen Vakilian

Department of Computer Science
University of Illinois at Urbana-Champaign
dpj@cs.uiuc.edu

Abstract
Today's shared-memory parallel programming models are complex and error-prone. While many parallel programs are intended to be deterministic, unanticipated thread interleavings can lead to subtle bugs and nondeterministic semantics. In this paper, we demonstrate that a practical type and effect system can simplify parallel programming by guaranteeing deterministic semantics with modular, compile-time type checking even in a rich, concurrent object-oriented language such as Java. We describe an object-oriented type and effect system that provides several new capabilities over previous systems for expressing deterministic parallel algorithms. We also describe a language called Deterministic Parallel Java (DPJ) that incorporates the new type system features, and we show that a core subset of DPJ is sound. We describe an experimental validation showing that DPJ can express a wide range of realistic parallel programs; that the new type system features are useful for such programs; and that the parallel programs exhibit good performance gains (coming close to or beating equivalent, nondeterministic multithreaded programs where those are available).

Categories and Subject Descriptors D.1.3 [Software]: Concurrent Programming—Parallel Programming; D.3.1 [Software]: Formal Definitions and Theory; D.3.2 [Software]: Language Classifications—Concurrent, distributed, and parallel languages; D.3.2 [Software]: Language Classifications—Object-oriented languages; D.3.3 [Software]: Language Constructs and Features—Concurrent Programming Structures

General Terms Languages, Verification, Performance
Keywords Determinism, deterministic parallelism, effects, effect systems, commutativity

1. Introduction
The advent of multicore processors demands parallel programming by mainstream programmers. The dominant model of concurrency today, multithreaded shared memory programming, is inherently complex due to the number of possible thread interleavings that can cause nondeterministic program behaviors. This nondeterminism causes subtle bugs: data races, atomicity violations, and deadlocks. The parallel programmer today prunes away the nondeterminism using constructs such as locks and semaphores, then debugs the program to eliminate the symptoms. This task is tedious, error prone, and extremely challenging even with good debugging tools.

The irony is that a vast number of computational algorithms (though not all) are in fact deterministic: a given input is always expected to produce the same output. Almost all scientific computing, encryption/decryption, sorting, compiler and program analysis, and processor simulation algorithms exhibit deterministic behavior. Today's parallel programming models force programmers to implement such algorithms in a nondeterministic notation and then convince themselves that the behavior will be deterministic.

By contrast, a deterministic-by-default programming model [9] can guarantee that any legal program produces the same externally visible results in all executions with a particular input unless nondeterministic behavior is explicitly requested by the programmer in disciplined ways. Such a model can make parallel application development and maintenance easier for several reasons. Programmers do not have to reason about notoriously subtle and difficult issues such as data races, deadlocks, and memory models. They can start with a sequential implementation and incrementally add parallelism, secure in the knowledge that the program behavior

---

*This work was supported by the National Science Foundation under grants CCF 07-02724 and CNS 07-20772, and by Intel, Microsoft and the University of Illinois through UPRC Illinois.
will remain unchanged. They can use familiar sequential tools for debugging and testing. Importantly, they can test an application only once for each input [19].

Unfortunately, while guaranteed determinism is available for some restricted styles of parallel programming (e.g., data parallel, or pure functional), it remains a challenging research problem to guarantee determinism for imperative, object-oriented languages such as Java, C++, and C#. In such languages, object references, aliasing, and updates to mutable state obscure the data dependencies between parts of a program, making it hard to prove that those dependences are respected by the program’s synchronization. This is a very important problem as many applications that need to be ported to multicore platforms are written in these languages.

We believe that a type and effect system [27, 26, 12, 30] is an important part of the solution to providing guaranteed deterministic semantics for imperative, object-oriented languages. A type and effect system (or effect system for short) allows the programmer to give names to distinct parts of the heap (we call them regions) and specify the kind of accesses to parts of the heap (e.g., read or write effects) in different parts of the program. The compiler can then check, using simple modular analysis, that all pairs of memory accesses either commute with each other (e.g., they are both reads, or they access disjoint parts of the heap) or are properly synchronized to ensure determinism. A robust type and effect system with minimal runtime checks is valuable because it enables checking at compile time rather than runtime, eliminates unnecessary runtime checks (thus leading to less overhead and/or less implementation complexity), and contributes to program understanding by showing where in the code parallelism is expressed — and where code must be rewritten to make parallelism available. Effect annotations can also provide an enforceable contract at interface boundaries, leading to greater modularity and composability of program components. An effect system can be supplemented with runtime speculation [23, 51, 38, 31, 50] or other runtime checks [43, 20, 47, 6] to enable greater expressivity.

In this paper, we develop a new type and effect system for expressing important patterns of deterministic parallelism in imperative, object-oriented programs. FX [33, 27] showed how to use regions and effects in limited ways for deterministic parallelism in a mostly functional language. Later work on object-oriented effects [26, 12, 30] and object ownership [16, 32, 14] introduced more sophisticated mechanisms for specifying effects. However, studying a wide range of realistic parallel algorithms has shown us that some significantly more powerful capabilities are needed for such algorithms. In particular, all of the existing work lacks general support for fundamental parallel patterns such as parallel updates on distinct fields of nested data structures, parallel array updates, in-place divide and conquer algorithms, and commutative parallel operations.

Our effect system can support all of the above capabilities, using several novel features. We introduce region path lists, or RPLs, which enable more flexible effect summaries, including effects on nested structures. RPLs also allow more flexible subtyping than previous work. We introduce an index-parameterized array type that allows references to provably distinct objects to be stored in an array while still permitting arbitrary aliasing of the objects through references outside the array. We are not aware of any statically checked type system that provides this capability. We define the notions of subarrays (i.e., one array that shares storage with another) and partition operations, that together enable in-place parallel divide and conquer operations on arrays. Subarrays and partitioning provide a natural object-oriented way to encode disjoint segments of arrays, in contrast to lower-level mechanisms like separation logic 35 that specify array index ranges directly. We also introduce an invocation effect, together with simple commutativity annotations, to permit the parallel invocation of operations that may actually interfere at the level of reads and writes, but still commute logically, i.e., produce the same final (logical) behavior. This mechanism supports concurrent data structures such as concurrent sets, hash maps, atomic counters, etc.

We have designed a language called Deterministic Parallel Java (DPJ) incorporating these features. DPJ is an extension to Java that enforces deterministic semantics via compile-time type checking. Because of the guaranteed deterministic semantics, existing Java code can be ported to DPJ incrementally. Furthermore, porting to DPJ will have minimal impact on program testing: developers can use the same tests and testing methodology for the ported parallel code as they had previously used for their sequential code.

The choice of Java for our work is not essential; similar extensions could be applied to other object-oriented languages, and we are currently developing a version of the language and compiler for C++. We are also exploring how to extend our type system and language to provide disciplined support for explicitly nondeterministic computations.

This paper makes the following contributions:

1. **Novel features.** We introduce a new region-based type and effect system with several novel features (RPLs, index-parameterized arrays, subarrays, and invocation effects) for expressing core parallel programming patterns in imperative languages. These features guarantee determinism at compile-time.

2. **Formal definition.** For a core subset of the type system, we have developed a formal definition of the static and dynamic semantics, and a detailed proof that our system allows sound static inference about noninterference of effects. We present an outline of the formal definition and proof in this paper. The full details are in an accompanying technical report [10] available via the Web [1].
3. **Language Definition.** We have designed a language called DPJ that incorporates the type and effect system into a modern O-O language (Java) in such a way that it supports the full flexibility of the sequential subset of Java, enables incremental porting of Java code to DPJ, and guarantees semantic equivalence between a DPJ program and its obvious sequential Java version. We have implemented a prototype compiler for DPJ that performs the necessary type checking and then maps parallelism down to the ForkJoinTask dynamic scheduling framework.

4. **Empirical evaluation.** We study six real-world parallel programs written in DPJ. This experience shows that DPJ can express a range of parallel programming patterns; that all the novel type system features are useful in real programs; and that the language is effective at achieving significant speedups on these codes on a commodity 24-core shared-memory processor. In fact, in 3 out of 6 codes, equivalent, manually parallelized versions written to use Java threads are available for comparison, and the DPJ versions come close to or beat the performance of the Java threads versions.

The rest of this paper proceeds as follows. Section 2 provides an overview of some basic features of DPJ, and Sections 3–5 explain the new features in the type system (RPLs, arrays, and commutativity annotations). Section 6 summarizes the formal results for a core subset of the language. Section 7 discusses our prototype implementation and evaluation of DPJ. Section 8 discusses related work, and Section 9 concludes.

2. **Basic Capabilities**

We begin by summarizing some basic capabilities of DPJ that are similar to previous work [33, 30, 26, 14, 15]. We refer to the example in Figure 1, which shows a simple binary tree with three nodes and a method `$\text{initTree}$` that writes into the `$\text{mass}$` fields of the left and right child nodes. As we describe more capabilities of DPJ, we will also expand upon this example to make it more realistic, e.g., supporting trees of arbitrary depth.

**Region names.** In DPJ, the programmer names regions to partition the heap, and writes method effect summaries stating what regions are read and written by each method. A field region declaration declares a new name `$r$` (called a field region name) that can be used as a region name. For example, line 2 declares names `$\text{Links}$`, `$L$`, and `$R$`, and these names are used as regions in lines 4 and 5. A field region name is associated with the static class in which it is declared; this fact allows us to reason soundly about effects without alias restrictions or interprocedural alias analysis. A field region name functions like an ordinary class member: it is inherited by subclasses, and outside the scope of its defining class, it must be appropriately qualified (e.g., `$\text{TreeNode}$.L`). A local region declaration is similar and declares a region name at local scope.

**Region parameters.** DPJ provides class and method region parameters that operate similarly to Java generic parameters. We declare region parameters with the keyword `region`, as shown in line 1, so that we can distinguish them from Java generic type parameters (which DPJ fully supports). When a region-parameterized class or method is used, region arguments must be provided to the parameters, as shown in lines 4–5. Region parameters enable us to create multiple instances of the same class with their data in different regions.

**Disjointness constraints.** To control aliasing of region parameters, the programmer may write a disjointness constraint [14] of the form `$P_1 \neq P_2$`, where `$P_1$` and `$P_2$` are parameters (or regions written with parameters; see Section 3) that are required to be disjoint. Disjointness of regions is fully explained in Section 3; in the case of simple names, it means the names must be different. The constraints are checked when instantiating the class or calling the method. If the disjointness constraints are violated, the compiler issues a warning.

**Partitioning the heap.** The programmer may place the keyword `region` in a field declaration, followed by the region.

```java
1. class TreeNode(region P) {
2.     region Links, L, R;
3.     double mass in P;
4.     TreeNode(L) left in Links;
5.     TreeNode(R) right in Links;
6.     void setMass(double mass) {
7.         begin {
8.             /* read Links write L */
9.             /* read Links write R */
10.            left mass = mass;
11.            right mass = mass;
12.         }
13.     }
14. }
15. }
```

**Figure 1.** Basic features of DPJ. Type and effect annotations are italicized. Note that method `initTree` (line 7) has no effect annotation, so it gets the default effect summary of "reads and writes the entire heap."

![Figure 2. Runtime heap typing from Figure 1](image)

**Figure 2.** Runtime heap typing from Figure 1.
as shown in lines 3–5. An operation on the field is treated as an operation on the region when specifying and checking effects. This effectively partitions the heap into regions. See Figure 2 for an illustration of the runtime heap typing, assuming the root node has been instantiated with Root.

Method effect summaries. Every method (including all constructors) must conservatively summarize its heap effects with an annotation of the form reads region-list writes region-list, as shown in line 6. Writes imply reads. When one method overrides another, the effects of the superclass method must contain the effects of the subclass method. For example, if a method specifies a writes effect, then all methods it overrides must specify that same writes effect. This constraint ensures that we can check effects soundly in the presence of polymorphic method invocation [30, 26]. The full DPJ language also includes effect variables [33], to support writing a subclass whose effects are unknown at the time of writing the superclass (e.g., in instantiating a library or framework class); however, we leave the discussion of effect variables to future work.

Effects on local variables need not be declared, because these effects are masked from the calling context. Nor must initialization effects inside a constructor body be declared, because the DPJ type and effect system ensures that no other task can access this until after the constructor returns. Read effects on final variables are also ignored, because those reads can never cause a conflict. A method or constructor with no externally visible heap effects may be declared pure.

To simplify programming and provide interoperability with legacy code, we adopt the rule that no annotation means "reads and writes the entire heap," as shown in Figure 1. This scheme allows ordinary sequential Java to work correctly, but it requires the programmer to add the annotations in order to introduce safe parallelism.

Expressing parallelism. DPJ provides two constructs for expressing parallelism, the cobegin block and the foreach loop. The cobegin block executes each statement in its body as a parallel task, as shown in lines 8–13. The foreach loop is used in conjunction with arrays and is described in Section 4.1.

Proving determinism. To type check the program in Figure 1, the compiler does the following. First, check that the summary writes P of method setMass (line 6) is correct (i.e., it covers all effect of the method). It is, because field mass is declared in region P (line 3), and there are no other effects. Second, check that the parallelism in lines 8–13 is safe. It is, because the effect of line 10 is reads Links writes L; the effect of line 12 is reads Links writes R; and Links, L, and R are distinct names. Notice that this analysis is entirely intraprocedural.

3. Region Path Lists (RPLs)

An important concept in effect systems is region nesting, which lets us partition the heap hierarchically so we can express that different computations are occurring on different parts of the heap. For example, to extend the code in Figure 1 to a tree of arbitrary depth, we need a tree of nested regions. As discussed in Section 4, we can also use nesting to express that two aggregate data structures (like arrays) are in distinct regions, and the components of those structures (like the cells of the arrays) are in distinct regions, each nested under the region containing the whole structure.

Effect systems that support nested regions are generally based on object ownership [16, 14] or use explicit declarations that one region is under another [30, 26]. As discussed below, we use a novel approach based on chains of elements called region path lists, or RPLs, that provides new capabilities for effect specification and subtyping.

3.1 Specifying Single Regions

The region path list (RPL) generalizes the notion of a simple region name r. Each RPL names a single region, or set of memory locations, on the heap. The set of all regions partitions the heap, i.e., each memory location lies in exactly one region. The regions are arranged in a tree with a special region Root as the root node. We say that one region is nested under (or simply under) another if the first is a descendant of the second in the tree. The tree structure guarantees that for any two distinct names r and r', the set of regions under r and the set of regions under r' have empty intersection, and we can use this guarantee to prove disjointness of memory accesses.

Syntactically, an RPL is a colon-separated list of names, called RPL elements, beginning with Root. Each element after Root is a declared region name r, for example, Root:A:B. As a shorthand, we can omit the leading Root. In particular, a bare name can be used as an RPL, as illustrated in Figure 1. The syntax of the RPL represents the nesting of region names: one RPL is under another if the second is a prefix of the first. For example, L:B is under L. We write R1 ≤ R2 if R1 is under R2.

We may also write a region parameter, instead of Root, at the head of an RPL, for example P:A, where P is a parameter. When a class with a region parameter is instantiated at runtime, the parameter is resolved to an RPL beginning with Root. Method region parameters are resolved similarly at method invocation time. Because a parameter P is always bound to the same RPL in a particular scope, we can make sound static inferences about parametric RPLs. For example, for all P, P:A ≤ P, and P:A ≠ P:B if and only if A ≠ B.

Figure 3 illustrates the use of region nesting and class region parameters to distinguish different fields as well as different objects. It extends the example from Figure 1 by

2 As noted in Section 2, this can be a package- or class-qualified name such as C::r; for simplicity, we use r throughout.
Figure 3. Extension of Figure 1 showing the use of region nesting and region parameters.

```plaintext
class TreeNode<region P> {  
  region Links, L, R, M, F;  
  double mass in P:M;  
  double force in P:F;  
  TreeNode<< left in Links;  
  TreeNode<< right in Links;  
  void initTree(double mass, double force) {  
    cobegin {  
      /* reads Links writes L:M */  
      left.mass = mass;  
      /* reads Links writes L:F */  
      left.force = force;  
      /* reads Links writes R:M */  
      right.mass = mass;  
      /* reads Links writes R:F */  
      right.force = force;  
    }  
  }  
}
```

Figure 4. Graphical depiction of the distinctions shown in Figure 3. The * denotes any sequence of RPL elements; this notation is explained further in Section 3.2.

adding a force field to the TreeNode class, and by making the initTree method (line 7) set the mass and force fields of the left and right child in four parallel statements in a cobegin block (lines 9–16).

To establish that the parallelism is safe (i.e., that lines 9–16 access disjoint locations), we place fields mass and force in distinct regions P:M and P:F, and the links left and right in a separate region Links (since they are only read). The parameter P appears in both regions and P is bound to different regions (L and R) for the left and right subtrees, because of the different instantiations of the parametric type TreeNode for the fields left and right. Because the names L and R used in the types are distinct, we can distinguish the effects on left (lines 10–12) from the effects on right (lines 14–16). And because the names M and F are distinct, we can distinguish the effects on the different fields within an object i.e., lines 10 vs. 14 and lines 12 vs. 16, from each other. Figure 4 shows this situation graphically.

3.2 Specifying Sets of Regions

Partially specified RPLs. To express recursive parallel algorithms, we must specify effects on sets of regions (e.g., "all regions under P"). To do this, we introduce partially specified RPLs. A partially specified RPL contains the symbol * ("star") as an RPL element, standing in for some unknown sequence of names. An RPL that contains no * is fully specified.

For example, consider the code shown in Figure 5. Here we are operating on the same TreeNode shown in Figs. 1 and 3, except that we have added (1) a Link field (line 7) that points to some other node in the tree and (2) a computeForces method (line 8) that recursively descends the tree. At each node, computeForces follows link to another node, reads the mass field of that node, computes the force between that node and this one, and stores the result in the force field of this node. This computation can safely be done in parallel on the subtrees at each level, because each call writes only the force field of this, and the operations on other nodes (through link) are all reads of the mass, which is distinct from force. To write this computation, we need to be able to say, for example, that line 13 writes only the left subtree, and does not touch the right subtree.

Distinctions from the left. In lines 11–15 of Figure 5, we need to distinguish the write to this.force (line 11) from the writes to the force fields in the subtrees (lines 13 and 15). We can use partially specified RPLs to do this. For example, line 8 says that computeForces may read all regions under Links and write all regions under P that end with F.

If RPLs R₁ and R₂ are the same in the first n places, they differ in place n + 1, and neither contains a * in the first n + 1 places, then (because the regions form a tree) the set of regions under R₁ and the set of regions under R₂ have empty intersection. In this case we say that R₁:* and R₂:* are disjoint, and we know that effects on these two RPLs are non-interfering. We call this a "distinction from the left," because we are using the distinctness of the names to the left of any star to infer that the region sets are non-intersecting. For example, a distinction from the left establishes that the region sets P:F; P:L:*; F; and P:B:*; F (shown in lines 10-15) are disjoint, because the RPLs all start with P and differ in the second place.
Distinctions from the right. Sometimes it is important to specify "all fields in any node of a tree." For example, in lines 10–15, we need to show that the reads of the mass fields are distinct from the writes to the force fields. We can make this kind of distinction by using different names after the star: if \( R_1 \) and \( R_2 \) differ in the \( n \)th place from the right, and neither contains a * in the first \( n \) places from the right, then a simple syntactic argument shows that their region sets are disjoint. We call this pattern a "distinction from the right," because the names that ensure distinctness appear to the right of any star. For example, in lines 10–15, we can distinguish the reads of \( *:M \) from the writes to \( P:L:*:F \) and \( P:R:*:F \).

More complicated patterns. More complicated RPL patterns like \( \text{Root}::A:*:B \) are supported by the type system. Although we do not expect that programmers will need to write such patterns, they sometimes arise via parameter substitution when the compiler is checking effects.

### 3.3 Subtyping and Type Casts

**Subtyping.** Partially specified RPLs are also useful for subtyping. For example, in Figure 5, we needed to type the field of a reference that could point to a \( \text{TreeNode}<P> \), for any binding to \( P \). With fully specified RPLs we cannot do this, because we cannot write a type to which we can assign both \( \text{TreeNode}<L> \) and \( \text{TreeNode}<R> \). The solution is to use a partially specified RPL in the type, e.g., \( \text{TreeNode}<> \), as shown in line 7 of Figure 5. Now we have a type that is flexible enough to allow the assignment, but retains soundness by explicitly saying that we do not know the actual region.

The subtyping rule is simple: \( C<R_2> \) is a subtype of \( C<R_3> \) if the set of regions denoted by \( R_2 \) is included in the set of regions denoted by \( R_3 \). We write \( R \subseteq R_2 \) to denote set inclusion for the corresponding sets of regions. If \( R_1 \) and \( R_2 \) are fully specified, then \( R_1 \subseteq R_2 \) implies \( R = R_2 \). Note that nesting and inclusion are related: \( R_1 \subseteq R_2 \) implies \( R_1 \subseteq R_2 * \). However, nesting alone does not imply inclusion of the corresponding sets. For example, \( A:B \subseteq A \), but \( A:B \not\subseteq A \), because \( A:B \) and \( A \) denote different regions. In Section 6 we discuss the rules for nesting, inclusion, and disjointness of RPLs more formally.

Figure 6 illustrates one possible heap typing resulting from the code in Figure 5. The DPJ typing discipline ensures the object graph restricted to the left and right references is a tree. However, the full object graph including the link references is more general and can have cycles, as illustrated in Figure 6. Note how our effect system is able to prove that the updates to different subtrees are distinct, even though (1) non-tree edges exist in the graph; and (2) those edges are followed to do possibly overlapping reads.

**Type casts.** DPJ allows any type cast that would be legal for the types obtained by erasing the region variables. This approach is sound if the region arguments are consistent. For example, given class \( B<r:region R> \) extends \( A<r> \), a cast from \( A<r> \) to \( B<r> \) is sound, because either the reference is \( B<r> \), or it is not any sort of \( B \), which will cause

![Figure 6. Heap typing from Figure 5. Reference values are shown by arrows; tree arrows are solid, and non-tree arrows are dashed. Notice that all arrows obey the subtyping rules.

4. **Arrays**

DPJ provides two novel capabilities for computing with arrays: index-parameterized arrays and subarrays. Index-parameterized arrays allow us to traverse an array of object references and safely update the objects in parallel, while subarrays allow us to dynamically partition an array into disjoint pieces, and give each piece to a parallel subtask.

4.1 **Index-Parameterized Arrays**

A basic capability of any language for deterministic parallelism is to operate on elements of an array in parallel. For a loop over an array of values, it is sufficient to prove that each iteration accesses a distinct array element (we call this a **unique traversal**). For a loop over an array of references to mutable objects, however, a unique traversal is not enough: we must also prove that any memory locations updated by following references in distinct array cells (possibly through a chain of references) are distinct. Proving this property is very hard in general, if assignments are allowed into reference cells of arrays. No previous effect system that we are aware of is able to ensure disjointness of updates by following references stored in arrays, and this severely limits the ability of those systems to express parallel algorithms.

In DPJ, we make use of the following insight:

**Insight 1.** We can define a special array type with the restriction that an object reference value \( o \) assigned to cell \( n \) (where \( n \) is a natural number constant) of such an array has a runtime type that is parameterized by \( n \). If accesses through cell \( n \) touch only region \( n \) (even by following a chain
class Body<region P> {
    region Link = P.F;
    double mass = P.M;
    double force = P.F;
    Body<<< link in Link;
    void computeForce() reads Link, #M writes P.F {
        force = (mass * link.mass) * X.DRAG;
    }
}

final Body<,>[][] bodies = new Body[,][,][][];
foreach (int i = 0, N) {
    /* reads [i] */
    bodies[i] = new Body<>()
    foreach (int i = 0, M) {
        /* reads [i], Link, #M writes [i].F */
        bodies[i].computeForce();
    }
}

Figure 7. Example using an index-parameterized array.

of references), then the accesses through different cells are
guaranteed to be disjoint.

We call such an array type an index-parameterized array.
To represent such arrays, we introduce two language constructs:

1. An array RPL element written [e], where e is an integer
   expression.

2. An index-parameterized array type that allows us to write
   the region and type of array cell e using the array RPL
   element [e]. For example, we can specify that cell e
   resides in region Root = [e] and has type <Root: [e]>
   At runtime, if e evaluates to a natural number n, then
   the static array RPL element [e] evaluates to the dynamic
   array RPL element [n].

The key point here is that we can distinguish C<[e1]>
from C<[e2]> if e1 and e2 always evaluate to unequal
values at runtime, just as we can distinguish C<r1> from C<r2>,
where r1 and r2 are declared names, as discussed in Section
3.1. Obviously, the compiler’s capability to distinguish
such types will be determined by its ability to prove the
independence of the symbolic expressions e1 and e2. This is possible
in many common cases, for the same reason that array
dependence analysis is effective in many, though not all,
cases [24]. The key benefit is that the type checker has then
proved the uniqueness of the target objects, which would not
follow from dependence analysis alone.

In DPL, the notation we use for index-parameterized
arrays is T[i] <R,.#i, where T is a type, R is an RPL,
#i declares a fresh integer variable i in scope over the type, and
[i] may appear as an array RPL element in T or R (or both).
This notation says that array cell e (where e is an integer
expression) has type T[i ← e] and is located in region
R[i ← e]. For example, C<x1:[L]>C<[x2:[z]>. #i specifies
an array such that cell e has type C<x1:[z]>. #i and resides
in region x2:[z]. If T itself is an array type, then nested
index variable declarations can appear in the type. However,
the most common case is a single-dimensional array, which
needs only one declaration. For that case, we provide a sim-
plified notation: the user may omit the #i and use an
underscore (_) as an implicitly declared variable. For example,
C<[x]>C<[y]. C is equivalent to C<[x]. #i)

Figure 7 shows an example, which is similar in spirit to
the Barnes-Hut force computation discussed in Section 7.
Lines 1–9 declare a class Body. Line 11 declares and creates
an index-parameterized array bodies with N cells, such that
each cell i resides in region [i] and points to an object of type
Body<,[i]>.

Figure 8 shows a sample heap typing, for some particular value n of N.

Lines 12–15 show a foreach loop that traverses the indices
i ∈ [0, n − 1] in parallel and initializes cell i with a new
object of type Body<,[i]>. The loop is noninterfering
because the type of bodies[i] resides in region [i], so distinct
iterations i and j write disjoint regions [i] and [j]. Lines 16–19 are similar,
except that the loop calls computeForce on each of the objects.
In iteration i of this loop, the effect of line 16 is reads [i], because it reads bodies[i].

To maintain soundness, we just need to enforce the invariant
that, at runtime, cell A[i] never points to an object of
type C<[j]> if i ̸= j. The compiler can enforce this
invariant through symbolic analysis, by requiring that if type
C<[e1]> is assigned to type C<[e2]> then e1 and e2 must
each evaluate to the same value at runtime; if it cannot
prove this fact, then it must conservatively disallow the
assignment. In many cases (as in the example above) the check
is straightforward.

Note that because of the typing rules, no two distinct
cells of an index-parameterized array can point to the same object.
However, it is perfectly legal to reach the same object by
following chains of references from distinct array cells, as
shown in Figure 8. In that case, in a parallel traversal over
the array, either the common object is not updated, in which
case the parallelism is safe; or a write effect on the same
region appears in two distinct iterations of a parallel loop, in
which case the compiler can catch the error.

Note also that while no two cells in an index-parameterized
array can alias, references may be freely shared with other
object that represents a contiguous subrange of the caller's array. We call this subrange a *subarray*. Notice that the DPJArray object does not replicate the underlying array; it stores only a reference to the underlying array, and the values of $S$ and $L$. The DPJArray object translates access to element $i$ into access to element $S + i$ of the underlying array. If $i < 0$ or $i \geq L$, then an array bounds exception is thrown, i.e., access through the subarray must stay within the specified segment of the original array.

Second, DPJ provides a class DPJPartition, representing an indexed collection of DPJArray objects, all of which point into mutually disjoint segments of the original array. To create a DPJPartition, the programmer passes a DPJArray object into the DPJPartition constructor, along with some arguments that say how to do the splitting. Lines 11–12 of Figure 9 show how to split the DPJArray $A$ at index $p$, and indicate that position $p$ is to be left out of the resulting disjoint segments. The programmer can access segment $i$ of the partition $\text{segs}$ by saying $\text{segs.get}(i)$, as shown in lines 15 and 17.

Third, to support recursive computations, we need a slight extension to the syntax of RPLs (Section 3). Notice that we cannot use a simple region name, like $r$, for the type of a partition segment, because different partitions can divide the same array in different ways. Instead, we allow a final local variable $z$ (including this) of class type to appear at the head of an RPL, for example $z\forall R$. The variable $z$ stands in for the object reference $o$ stored into the variable at runtime, which is the actual region. Using the object reference as a region ensures that different partitions get different regions, and making the variable final ensures that it always refers to the same region.

We make these "z regions" into a tree as follows. If $z$'s type is $C<R, \ldots>$, then $z$ is nested under $R$; the first region parameter of a class functions like the *owner parameter* in an object ownership system [18, 16]. In the particular case of DPJPartition, if the type of $z$ is DPJPartition$<R>$, then the type of $z.get(i)$ is $z[i]:*,[<\ldots],$ where $z \leq R$. Internally, the get method uses a type cast to generate a DPJArray of type $z[i]:*$ that points into the underlying array. The type cast is not sound within the type system, but it is hidden from the user code in such a way that all well-typed uses of DPJPartition are noninterfering.

In Figure 9, the sequence of recursive sort calls creates a tree of QSort objects, each in its own region. The cobegin in lines 13–17 is safe because DPJPartition guarantees that the segments $\text{segs.get}(0)$ and $\text{segs.get}(1)$ passed into the recursive parallel sort calls are disjoint. In the user code, the compiler uses the type and effect annotations to prove noninterference as follows. First, from the type of QSort and the declared effect of sort (line 4), the compiler determines that the effects of lines 15 and 17 are $\text{writes segs[0]}:*$ and $\text{writes segs[1]}:*$, as shown. Second, the regions $\text{segs[0]}:*$ and $\text{segs[1]}:*$ are disjoint.

4.2 Subarrays

A familiar pattern for writing divide and conquer recursion is to partition an array into two or more disjoint pieces and give each array to a subtask. For example, Figure 9 shows a standard implementation of quicksort, which divides the array in two at each recursive step, then works in parallel on the halves. DPJ supports this pattern with three novel features, which we illustrate with the quicksort example.

First, DPJ provides a class DPJArray that wraps an ordinary Java array and provides a view into a contiguous segment of it, parameterized by start position $S$ and length $L$. In Figure 9, the QSort constructor (line 3) takes a DPJArray
joint, by a distinction from the left (Section 3.2). Finally, the effect writes P:* in line 4 correctly summarizes the effects of sort, because lines 6 and 9 write P, lines 15 and 17 write under ssga, and ssga is under P, as explained above.

Notice that DPJPartition can create multiple error messages to overlapping data with different regions in the types. Thus, there is potential for unsoundness here if we are not careful. To make this work, we must do two things. First, if z_1 and z_2 represent different partitions of the same array, then z_1.get(0) and z_2.get(1) could overlap. Therefore, we must not treat them as disjoint. This is why we put * at the end of the type z[:i]* of z.get(i); otherwise we could incorrectly distinguish z_1[0] from z_2[1], using a distinction from the right. Second, if z has type DPJPartition<R>, then z.get(i) has type DPJArray<z[:,:]> and points into a DPJArray<R>. Therefore, we must not treat z[:i]* as disjoint from R.

Here, we simply do not include this distinction in our type system. All we say is that z[:i]* \leq R. See Section 6.3 and Appendix C.2 for further discussion of the disjointness rules in our type system.

5. Commutativity Annotations

Sometimes to express parallelism we need to look at interference in terms of higher-level operations than read and write [29]. For example, insertions into a concurrent Set can go in parallel and preserve determinism even though the order of interfering reads and writes inside the Set implementation is nondeterministic. Another example is computing connected components of a graph in parallel.

In DPI, we address this problem by adding two features. First, classes may contain declarations of the form m commuteswith m', where m and m' are method names, indicating that any pair of invocations of the named methods may be safely done in parallel, regardless of the read and write effects of the methods. See Figure 10(a). In effect, the commuteswith annotation says that (1) the two invocations are atomic with respect to each other, i.e., the result will be as if one occurred and then the other; and (2) either order of invocation produces the same result.

The commutativity property itself is not checked by the compiler; we must rely on other forms of checking (e.g., more complex program logic [52] or static analysis [42, 4]) to ensure that methods declared to be commutative are really commutative. In practice, we anticipate that commuteswith will be used mostly by library and framework code that is written by experienced programmers and extensively tested. Our effect system does guarantee deterministic results for an application using a commutative operation, assuming that the operation declared commutative is indeed commutative.

Second, our effect system provides a new invocation effect of the form invokes m with E. This effect records that an invocation of method m occurred with underlying effects E. The type system needs this information to represent and check effects soundly in the presence of commutativity annotations: for example, in line 4 of Fig. 10(b), the compiler needs to record that add was invoked there (so it can disregard the effects of other add invocations) and that the underlying effect of the method was writes R (so it can verify that there are no other interfering effects, e.g., reads or writes of R, in the invoking code).

When there are one or more intervening method calls between a foreach loop and a commutative operation, it may also be necessary for a method effect summary in the program text to specify that an invocation occurred inside the method. For example, in Figure 10(c), the add method is called through a wrapper object. We could have correctly specified the effect of Adder.add as writes P, but this would hide from the compiler the fact that Adder.add commutes with itself. Of course we could use commuteswith for Adder.add, but this is highly unsatisfactory: it just propagates the unchecked commutativity annotation out through the call chain in the application code. The solution is to specify the invocation effect invokes IntSet.add with writes P, as shown.

Notice that the programmer-specified invocation effect exposes an internal implementation detail (i.e., that a particular method was invoked) at a method interface. However, we believe that such exposure will be rare. In most cases, the effect invokes C::m with E will be conservatively summarized as E (Section 6.1 gives the formal rules for covering effects). The invocation effect will only be used for cases where a commutative method is invoked, and the commutativity information needs to be exposed to the caller. We believe these cases will generally be confined to high-level public API methods, such as Set.add in the example given in Figure 10.
6. The Core DPJ Type System

We have formalized a subset of DPJ, called Core DPJ. To make the presentation more tractable and to focus attention on the important aspects of the language, we make the following simplifications:

1. We present a simple expression-based language, omitting more complicated aspects of the real language such as statements and control flow.

2. Our language has classes and objects, but no inheritance.

3. Region names are declared at global scope, instead of at class scope. Every class has one region parameter, and every method has one parameter.

4. To avoid dealing with integer variables and expressions, we require that array indices are natural number literals.

Removing the first simplification adds complexity but raises no significant technical issues. Adding inheritance raises standard issues for formalizing an object-oriented language. We omit those here in order to focus on the novel aspects of our system, but we describe them in [10]. Removing simplifications 3 and 4 is mostly a matter of bookkeeping. To handle arrays in the full language, we need to prove equivalence and non-equivalence of array index expressions, but this is a standard compiler capability.

We have chosen to make Core DPJ a sequential language, in order to focus on our mechanisms for expressing effects and noninterference. In Section 6.4, we discuss how to extend the formalism to model the cobegin and foreach constructs of DPJ.

6.1 Syntax and Static Semantics

Figure 11 defines the syntax of Core DPJ. The syntax consists of the key elements described in the previous sections (RPLs, effects, and commutativity annotations) hung upon a toy language that is sufficient to illustrate the features yet reasonable to formalize. A program consists of a number of region declarations, a number of class declarations, and an expression to evaluate. Class definitions are similar to Java’s, with the restrictions noted above.

![Figure 11. Core DPJ syntax. C, P, f, m, x, r, and i are identifiers, and n is a natural number. R_f denotes a fully specified RPL (i.e., containing no *)](image)

![Figure 12. Core DPJ type judgments. We extend the judgments to groups of things (e.g., \( \Gamma \vdash \text{field} \)) in the obvious way.](image)

We define the static semantics of Core DPJ with the judgments stated in Figure 12. The judgments are defined with respect to an environment \( \Gamma \), where each element of \( \Gamma \) is one of the following:

- A binding \( z \rightarrow T \) stating that variable \( z \) has type \( T \).
  These elements come into scope when a new variable (let variable or formal parameter) is introduced.

- A constraint \( P \rightarrow R \) stating that region parameter \( P \) is in scope and included in region \( R \). These elements come into scope when we capture the type of a variable used for an invocation (see the discussion of expression typing judgments below).

- An integer variable \( i \). These elements come into scope when we are evaluating an array expression or a new array expression.

The formal rules for making the judgments are stated in full in Appendix A. Below we briefly discuss each of the five groups of judgments.

**Programs.** These judgments state that a program and its top-level components (classes, methods, etc.) are valid. Most rules just require that the component’s components are valid in the surrounding environment. The rule for valid method definitions (METHOD) requires that the method body’s type and effect are a subtype and subeffect of the return type and declared effect. These constraints ensure that we can use the method declaration to reason soundly about a method’s return type and effect when we are typing method invocation expressions.

**RPLs.** These judgments define validity, nesting, and inclusion of RPLs. Most rules are a straightforward formal translation of the relations that we described informally in Section 3.2. The key rule states that if \( R \) is under \( R' \) in some environment, then \( R \) is included in \( R' \) in that environment:
Types. These define when one type is a subtype of another. The class subtyping rule is just the formal statement of the rule we described informally in Section 3.3:

(INCLUDE-STAR) \[
\Gamma \vdash R \subseteq R' \quad \Gamma \vdash R \subseteq R' : * 
\]

The array subtyping rule is similar:

(Subtype-Array) \[
\Gamma \cup \{i\} \vdash R \subseteq R' \quad i = \mathbf{T} \quad \Gamma \vdash T \sqsubseteq T' \quad \Gamma \vdash \overline{T} \subseteq \overline{T}' \quad \overline{R} \sqsubseteq \overline{R}' 
\]

Here \(\equiv\) means identity of element types up to the names of integer variables \(i\). More flexible element subtyping is not possible without sacrificing soundness. We could allow unsound assignments and check for them at runtime (as Java does for class subtyping of array elements), but this would require that we retain the class region binding information at runtime.

Effects. These judgments define when an effect is valid, and when one effect is a subeffect of another. Intuitively, "\(E\) is a subeffect of \(E'\)" means that \(E'\) contains all the effects of \(E\), i.e., we can use \(E'\) as a (possibly conservative) summary of \(E\). The rules for reads, writes, and effect unions are standard [16, 33], but there are two new rules for invocation effects. First, if \(E'\) covers \(E\), then an invocation of some method with \(E'\) covers an invocation of the same method with \(E\):

(Se-Invokes-1) \[
\Gamma \vdash E \subseteq E' \quad \Gamma \vdash \text{invokes } C.m \text{ with } E' \rightarrow \Gamma \vdash \text{invokes } C.m \text{ with } E
\]

Second, we can conservatively summarize the effect \(\text{invokes } C.m \text{ with } E\) as just \(E\):

(Se-Invokes-2) \[
\Gamma \vdash \text{invokes } C.m \text{ with } E' \rightarrow \Gamma \vdash \text{invokes } C.m \text{ with } E
\]

Expressions. These judgments tell us how to compute the type and effect of an expression. They also ensure that the types of component expressions (for example at assignments and method parameter bindings) match in a way that guarantees soundness. The rules for field and array access and assignment, variable lookup, and new classes and arrays are straightforward. For the rule for let, \(x = e\) in \(e'\), we type \(x\) to the type of \(e\), and type \(e'\). If \(x\) appears in the type or effect of \(e'\), we replace it with \(R::*\) to generate a type and effect for the whole expression that is valid in the outer scope.

In the rule for method invocation \((\text{Invoke})\), we translate the type \(T_p\) of the method formal parameter to the current context by creating a fresh region parameter \(P\) included in the region \(R\) of \(z\)'s type. This technique is similar to how Java handles the capture of a generic wildcard. Note that simply substituting \(R\) for \(\text{param}(C)\) in translating \(T_p\) would not be sound; see [10] for an explanation and an example.

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPLs</td>
<td>(dR)</td>
<td>Root (\cdot)</td>
</tr>
<tr>
<td>Types</td>
<td>(dT)</td>
<td>(C::R)</td>
</tr>
<tr>
<td>Effects</td>
<td>(dE)</td>
<td>{ reads (dR)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>invokes (C.m) with (dE)</td>
</tr>
</tbody>
</table>

![Figure 13. Dynamic syntax of Core DPJ.](image)

\(dR\) denotes a fully-specified dynamic RPL (i.e., containing no \(*\)).

We also check that the actual argument type is a subtype of the declared formal parameter type, and we report the invocation of the method with its declared effect.

6.2 Dynamic Semantics

The syntax for entities appearing in the dynamic semantics is shown in Figure 13. At runtime, we have dynamic regions \((dR)\), dynamic types \((dT)\) and dynamic effects \((dE)\), corresponding to static regions \((R)\), types \((T)\) and effects \((E)\) respectively. Dynamic regions and effects are not recorded in a real execution, but they are passed through the execution state so we can formulate and prove soundness results [16]. We also have object references \(o\), which are the actual values computed during the execution.

The dynamic execution state consists of (1) a heap \(H\), which is a function taking values to objects; and (2) a dynamic environment \(d\), which is a set of elements of the form \(z \mapsto o\) (variable \(z\) is bound to value \(o\)) or \(P \mapsto dR\) (region parameter \(P\) is bound to region \(dR\)). \(d\) defines a natural substitution on RPLs, where we replace the variables with values and the region parameters with regions as specified in the environment. We denote this substitution on RPL \(R\) as \(d.R\), and we extend this notation to types and effects in the obvious way. Notice that we get the syntax of Figure 13 by applying the substitution \(d\) to the syntax of Figure 11.

An object is a partial function taking field names to object references. If the function is undefined on all field names, then we say it is a \textbf{null object}. We use null objects because we need to track the actual types of null references to establish soundness. Since the actual implementation does not need to do this tracking, it can just use the single value \texttt{null}. Every object reference \(o \in \text{Dom}(H)\) has a type, determined when the object is created, and we write \(H \rightharpoonup o : dT\) to mean that the reference \(o\) has type \(dT\) with respect to heap \(H\).

We write the evaluation rules in large-step semantics notation, using the following evaluation function:

\[(e, d, H) \rightarrow (o, H', dE),\]

where \(e\) is an expression to evaluate, \(dT\) and \(H\) give the dynamic context for evaluation, \(o\) is the result of the evaluation, \(H'\) is the updated heap, and \(dE\) represents the effects of the evaluation. A program evaluates to reference \(o\) with heap \(H\) and effect \(dE\) if its main expression is \(e\) and \((e, \emptyset, \emptyset) \rightarrow (o, H, dE)\).

Section B of the Appendix states the rules for program evaluation. The rules are standard for an imperative language, except that we record read effects in DYN-FIELD-
ACCESS and DYN-ARRAY-ACCESS and write effects in DYN-FIELD-ASSIGN and DYN-ARRAY-ASSIGN. Rules DYN-LET and DYN-INVOKE accumulate the effects of the component expressions. Note that when we evaluate \( \text{new } T \) we eliminate any \( * \) from \( T \) in the dynamic type of the new reference, e.g., \( \text{new } C<\text{Root}:: \) is the same type as \( C<\text{Root}:: \); this rule ensures that all object fields are allocated in fully specified regions. This rule is sound for the same reason that assigning \( C<\text{Root}:: \) to a variable of type \( C<\text{Root}:: \) is sound.

6.3 Soundness

Our key soundness result is that we can define and check a static property of noninterference of effect between expressions in the language, such that static noninterference implies dynamic noninterference. Appendix C states the major steps of the proof in formal terms. We divide the steps into three groups: type and effect preservation (Section C.1), disjointness (Section C.2), and noninterference of effect (Section C.3). We provide further explanation and a full proof in our technical report [10].

Type and effect preservation. In Section C.1, we assert some preliminary definitions and the preservation result. A dynamic environment \( dT \) is valid (Definition 1) if the types and RPLs appearing on the right of its bindings are valid, and it is internally consistent. A heap \( H \) is valid (Definition 2) if the reference stored in every object field or array cell of \( H \) is consistent with the declared type of the field or cell, translated to \( dT \). A dynamic environment \( dT \) instantiates a static environment \( \Gamma \) (Definition 3) if the bindings to variables in \( dT \) are consistent with the bindings to the corresponding variables in \( \Gamma \), after translation to \( dT \).

Theorem 1 establishes that we can use the static types and effects (Section 6.1) to reason soundly about dynamic types and effects (Section 6.2). It states that if we type an expression \( e \) in environment \( \Gamma \), and we evaluate \( e \) in dynamic environment \( dT \), where \( dT \) instantiates \( \Gamma \), then (a) the evaluation takes a valid heap to a valid heap; (b) the static type of \( e \) bounds the dynamic type of the value \( o \) that results from the evaluation; and (c) the static effect of \( e \) bounds the dynamic effect that results from the evaluation.

Disjoint RPLs. In Section C.2, we formally define a disjointness relation on pairs of RPLs (\( \Gamma \vdash R \neq R' \)). The relation formalizes distinctions from the left and right, as discussed informally in Section 3.2. Definition 4 formally expresses how to interpret a dynamic RPL as a set of fully-specified RPLs (i.e., regions). Definition 5 shows how to associate every object field and array cell with a region of the heap.

Proposition 1 states that disjoint RPLs imply disjoint sets of fully-specified regions, i.e., disjoint sets of locations. Proposition 2 states that at runtime, disjoint fully-specified regions imply disjoint locations.

Noninterference. In Section C.3, we formally define a noninterference relation on pairs of static effects (\( \Gamma \vdash E \neq E' \)). The rules express four basic facts: (1) reads commute with reads; (2) writes commute with reads or writes if the regions are disjoint; (3) invocations commute with other effects if the underlying effects are disjoint; and (4) two invocations commute if the methods are declared to commute, regardless of interference between the underlying effects.

Theorem 2 expresses the main soundness property of Core DPJ, which is that the execution order of noninterfering expressions does not matter. It states that in a well-typed program, if \( e \) and \( e' \) are expressions with types \( T \) and \( T' \) and effects \( E \) and \( E' \), and \( E \) and \( E' \) are noninterfering, then either order of evaluating \( e \) and \( e' \) produces the same values \( o \) and \( o' \), the same effects \( dE \) and \( dE' \), and the same final heap \( H \).

The claim is true for dynamic effects from the commutativity of reads, the disjointness results of Section C.2, and the assumed correctness of the commutativity specifications for methods. The claim is true for static effects by the type and effect preservation property above. See [10] for the formal proof.

6.4 Deterministic Parallelism

As discussed in Sections 2 and 4, the actual DPJ language includes \( \text{foreach} \) for parallel loops and \( \text{cobegin} \) for a block of parallel statements. We briefly discuss how to extend the formalism to model these constructs.

We can easily simulate \( \text{cobegin} \) by adding a parallel composition operator \( e/e' \), which says to execute \( e \) and \( e' \) in the same environment, in an unspecified order, with an implicit join at the end of the execution. We can simulate \( \text{foreach} \) by allowing an induction variable \( i \) to appear in expressions inside the scope of a \( \text{foreach} \), mapping \( i \) to \( n \) over the index range of the \( \text{foreach} \), and evaluating all \( e_n \) in unspecified order. In both cases we can extend the static typing rules to say that for any pair of expressions \( e \) and \( e' \) as to which the order of execution is unspecified, then the effects of \( e \) and \( e' \) must be noninterfering.

It follows directly from Theorem 2 that parallel composition of noninterfering expressions produces the same result as sequential composition of those expressions. This guarantees determinism of execution regardless of the order of parallel execution. The formalization of this property is straightforward, and we omit it from our technical report.

7. Evaluation

We have carried out a preliminary evaluation of the language and type system features presented in this paper. Our evaluation addressed the following questions:

• Expressiveness. Can the type system express important parallel algorithms on object-oriented data structures? When does it fail to capture parallelism and why?

• Coverage. Are each of the \( \text{new} \) features in the DPJ type system important to express one or more of these algorithms?
• **Performance.** For each of the algorithms, what increase in performance is realized in practice? This is a quantitative measure of how much parallelism the type system can express for each algorithm (note that the runtime overheads introduced by DPJ are negligible).

To do the evaluation, we extended Sun’s Javac compiler so that it compiles DPJ into ordinary Java source. We built a runtime system for DPJ using the ForkJoinTask framework that will be added to the java.util.concurrent standard library in Java 1.7 [1]. ForkJoinTask supports dynamic scheduling of lightweight parallel tasks, using a work-stealing scheduler similar to that in Cilk [8]. The DPJ compiler automatically translates foreach to a recursive computation that successively divides the iteration space, to a depth that is tunable by the programmer, and it translates a compute block into one task for every statement. Code using ForkJoinTask is compatible with Java threads so an existing multithreaded Java program can be incrementally ported to DPJ. Such code may still have some guarantees, e.g., the DPJ portions will be guaranteed deterministic if the explicitly threaded and DPJ portions are separate phases that do not run concurrently.

Using the DPJ compiler, we studied the following programs: Parallel merge sort, two codes from the Java Grande parallel benchmark suite (a Monte Carlo financial simulation and IDEA encryption), the force computation from the Barnes-Hut n-body simulation and IDEA encryption), the force computation from the Barnes-Hut n-body simulation [45], k-means clustering from the STAMP benchmarks [34], and a tree-based collision detection algorithm from a large, real-world open source game engine called JMonkey (we refer to this algorithm as Collision Tree). For all the codes, we began with a sequential version and modified it to add the DPJ type annotations. The Java Grande benchmarks are explicitly parallel versions using Java threads (along with equivalent sequential versions), and we compared DPJ’s performance against those. We also wrote and carefully tuned the Barnes-Hut force computation using Java threads as part of understanding performance issues in the code, so we could compare Java and DPJ for that one as well.

### 7.1 A Realistic Example

We use the Barnes-Hut force computation to show how to write a realistic parallel program in DPJ. Figure 14 shows a simplified version of this code. The main simplification is that the Vector objects are immutable, with final fields (so there are no effects on these objects), whereas our actual implementation uses mutable objects. The class Node represents an abstract tree node containing a mass and position. The mass and position represent the actual mass and position of a body (at a leaf) or the center of mass of a subtree (at an inner node). The Node class has two subclasses: InnerNode, representing an inner node of the tree, and storing an array of children; and Body, representing the body data stored at the leaves, and storing a force. The Tree class stores the tree,

```java
/* Abstract class for tree nodes */
abstract class Node {
  ... // Region...
}
/* Inner node of the tree */
class InnerNode extends Node {
  region Children;
  Node[0] x children; children in R:Children;
}
/* Leaf node of the tree */
class Body extends Node { /* Region for force */
  Vector force in R:Force; /* Force on this body */
}
/* Compute force of entire subtree on this body */
Vector computeForce(Node x x subtree) reads R:Children, R:MP
  ... }
/* Barnes-Hut tree */
class Tree extends Node { /* Region for tree */
  region Tree;
  Node root in R:Tree; /* Root */
  Body[N:1] x x bodies in R:Tree; /* Leaves */
}
/* Compute forces on all bodies */
void computeForce() writes R: ( /* foreach(int i in 0, bodies.length) */
  /read R.Tree, R:Node.Children, R:[1],
  R:Node MP writes R:[1].Node.force =
  bodies[i].force = bodies[i].computeForce(root);
}
}
```

*Figure 14.* Using DPJ to write the Barnes-Hut force computation.

Figure 14 shows a simplified version of this code. The main simplification is that the Vector objects are immutable with final fields (so there are no effects on these objects), whereas our actual implementation uses mutable objects. The class Node represents an abstract tree node containing a mass and position. The mass and position represent the actual mass and position of a body (at a leaf) or the center of mass of a subtree (at an inner node). The Node class has two subclasses: InnerNode, representing an inner node of the tree, and storing an array of children; and Body, representing the body data stored at the leaves, and storing a force. The Tree class stores the tree,

The method Tree.computeForce does the force computation by traversing the array of bodies and calling the method Body.computeForce on each one, to compute the force between the body this and subtree. If subtree is a body, or is sufficiently far away that it can be approximated as a point mass, then Body.computeForce computes and returns the pairwise interaction between the nodes. Otherwise, it recursively calls computeForce on the children of subtree, and accumulates the result.

We use a region parameter on the node classes to distinguish instances of these nodes. Class Tree uses the parameters to create an index-parameterized array of references to distinct body objects; the parallel loop in computeForces iterates over this array. This allows distinctions from the left for operations on bodies[1] (Section 5). We also use distinct region names within each class (in particular, for the force, masses, and positions, and the children array) to enable distinctions from the right.

The key fact is that, from the effect summary in line 21 and the code in line 35, the compiler infers the effects shown in lines 33–34. Using distinctions from the left and right, the compiler can now prove that (1) the updates are distinct for distinct iterations of the foreach; and (2) all the updates are distinct from the reads. Notice also how the nested RPLs
allow us to describe the entire effect of computeForce as writes R*: That is, to the outside world, computeForce just writes under the region parameter of Tree. Thus with careful use of RPLs, we can enforce a kind of encapsulation of effects, which is important for modular software design.

7.2 Expressiveness and Coverage

We used DPJ to express all available parallelism (except for vector parallelism, which we do not consider here) for Merge Sort, Monte Carlo, IDEA, K-Means, and Collision Tree. For Barnes-Hut, the overall program includes four major phases in each time step: tree building; center-of-mass computation; force calculations; and position calculations. Expressing the force, center of mass, and position calculations is straightforward, but we studied only the force computation (the dominant part of the overall computation) for this work. DPJ can also express the tree-building phase, but we would have to use a divide-and-conquer approach, instead of inserting bodies from the root via "hand-over-hand locking," as in in [45].

Briefly, we parallelized each of the codes as follows. MergeSort uses subarrays (Section 4.2) to perform in-place parallel divide and conquer operations for both merge and sort, switching to sequential merge and sort for subproblems below a certain size. Monte Carlo uses index-parameterized arrays (Section 4.1) to generate an array of tasks and compute an array of results, followed by commutativity annotations (Section 5) to update to globally shared data inside a reduction loop. IDEA uses subarrays to divide the input array into disjoint pieces, then uses foreach to operate on each of the pieces. Section 7.1 describes our parallel Barnes-Hut force computation. Collision Tree recursively walks two trees, reading the trees and collecting a list of intersecting triangles. At each node, a separate triangle list is computed in parallel for each subtree, and then the lists are merged. Our implementation uses method-local regions to distinguish the writes to the left and right subtree list. K-Means uses commutativity annotations to perform simultaneous reductions, one for each cluster. Table 1 summarizes the novel DPJ capabilities used for each code.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Sort</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Monte Carlo</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>IDEA</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>Barnes-Hut</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Collision Tree</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>K-Means</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
</tbody>
</table>

Table 1. Capabilities Used In The Benchmarks

1. Index-parameterized array; 2. Distinctions from the left; 3. Distinctions from the right; 4. Recursive subranges; 5. Commutativity annotations.

Our evaluation and experience showed some interesting limitations of the current language design. To achieve good cache performance in Barnes-Hut, the bodies must be reordered according to their proximity in space on each time step [45]. As discussed in Section 7.1, we use an index-

<table>
<thead>
<tr>
<th>Num</th>
<th>Monte Carlo</th>
<th>IDEA</th>
<th>Barnes Hut</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>DPJ Java</td>
<td>DPJ Java</td>
<td>DPJ Java</td>
</tr>
<tr>
<td>2</td>
<td>1.90</td>
<td>1.95</td>
<td>1.99</td>
</tr>
<tr>
<td>3</td>
<td>2.82</td>
<td>2.88</td>
<td>2.97</td>
</tr>
<tr>
<td>4</td>
<td>3.56</td>
<td>3.80</td>
<td>3.91</td>
</tr>
<tr>
<td>7</td>
<td>5.53</td>
<td>6.40</td>
<td>6.79</td>
</tr>
<tr>
<td>12</td>
<td>8.01</td>
<td>9.99</td>
<td>11.04</td>
</tr>
<tr>
<td>17</td>
<td>10.02</td>
<td>12.70</td>
<td>14.90</td>
</tr>
<tr>
<td>22</td>
<td>11.50</td>
<td>18.70</td>
<td>17.79</td>
</tr>
</tbody>
</table>

Table 2. Comparison of DPJ vs. Java threads performance for Monte Carlo, IDEA encryption, and Barnes Hut.

parameterized array to update the bodies in parallel. As discussed in Section 4.1, this requires that we copy each body with the new destination regions at the point of re-insertion. As future work, we believe we can ease this restriction by adding a mechanism for disjointness checking at runtime.

7.3 Performance

We measured the performance of each of the benchmarks on a Dell R900 multiprocessor running Red Hat Linux with 24 cores, comprising four six-core Xeon processors, and a total of 48GB of main memory. For each data point, we took the minimum of five runs on an idle machine.

We studied multiple inputs for each of the benchmarks and also experimented with different limits for recursive codes. We present results for the inputs and parameter values that show the best performance, since our main aim is to evaluate how well DPJ can express the parallelism in these codes. The sensitivity of the parallelism to input size and recursive limit parameters is a property of the algorithm and not a consequence of using DPJ.

Figure 15 presents the speedups of the six programs for \( p \in \{1, 2, 3, 4, 7, 12, 17, 22\} \) processors. All speedups are relative to an equivalent sequential version of the program, with no DPJ or other multithreaded runtime overheads. All six codes showed moderate to good scalability for all values of \( p \). Barnes-Hut and Merge Sort showed near-ideal performance scalability, with Barnes-Hut showing a superlinear increase for \( p = 22 \) due to cache effects.

Notably, as shown in Table 2, for the three codes where we have manually parallelized Java threads versions available, the DPJ versions achieved speedups close to (IDEA and Barnes Hut), or better than (Monte Carlo), the Java versions, for the same inputs on the same machines. We believe the Java threads codes are all reasonably well tuned; the two Java Grande benchmarks were tuned by the original authors and the Barnes Hut code was tuned by us. The manually parallelized Monte Carlo code exhibited a similar leveling off in speedup as the DPJ version did beyond about 7 cores because both have a significant sequential component that makes copies of a large array for each parallel task. Overall, in all three programs, DPJ is able to express the available parallelism as efficiently as a lower-level hand coded parallel programming model that provides no guarantees of determinism or even race-freedom.
Figure 15. Speedups. Numbers in legend are input sizes.

Our experience so far has shown us that DPJ itself can be very efficient, even though both the compiler and runtime are preliminary. In particular (except for very small runtime costs for the dynamic partitioning mechanism for subarrays), our type system requires no runtime checks or speculation and therefore adds negligible runtime overhead for achieving determinism. On the other hand, it is possible that the type system may constrain algorithmic design choices. The limitation on reordering the array of bodies in Barnes-Hut, explained in Section 7.2, is one such example.

7.4 Porting Effort

Table 3 shows the number of source lines changed and the number of annotations, relative to the program size. Program size is given in non-blank, non-comment lines of source code, counted by sloccount. The next column shows how many LOC were changed when annotating. The last four columns show (1) the number of declarations using the region keyword (i.e., field regions, local regions, and region parameters); (2) the number of RPLs appearing as arguments to in, types, methods, and effect summaries; (3) the number of method effect summaries, counting reads and writes separately; and (4) the number of commutativity annotations. As the table shows, the fraction of lines of code changed was not large, averaging 10.7% of the original lines. Most of the changed lines were due to writing RPL arguments when instantiating types (represented in column four), followed by writing method effect summaries (column five).

More importantly, we believe that the overall effort of writing, testing, and debugging a program with any parallel programming model is dominated by the time required to understand the parallelism and sharing patterns (including aliases), and to debug the parallel code. The regions and effects in DPJ provide concrete guidance to the programmer on how to reason about parallelism and sharing. Once the programmer understands the sharing patterns, he or she explicitly documents them in the code through region and effect annotations, so other programmers can gain the benefit of his or her understanding.

Further, programming tools can alleviate the burden of writing annotations. We have developed an interactive porting tool, DPJIZER [49], that infers many of these annotations, using iterative constraint solving over the whole program. DPJIZER is implemented as an Eclipse plugin and correctly infers method effect summaries for a program that is already annotated with region information. We are currently extending DPJIZER to infer RPLs, assuming that the programmer declares the regions.

In addition, a good set of defaults can further reduce the amount of manually written annotations. For example, if the programmer does not annotate a class field, its default region could be the RPL default-parameter:field-name. This default distinguishes both instances of the same class and fields within a class. The programmer can override the defaults if she needs further refinements.

Table 3. Annotation counts for the case studies.

<table>
<thead>
<tr>
<th>Program</th>
<th>Total LOC</th>
<th>Annotated LOC</th>
<th>Region Decls</th>
<th>RPLs</th>
<th>Summary</th>
<th>Comm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MergeSort</td>
<td>295</td>
<td>38 (12.9%)</td>
<td>15</td>
<td>41</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>Monte Carlo</td>
<td>2877</td>
<td>220 (7.6%)</td>
<td>13</td>
<td>301</td>
<td>161</td>
<td>1</td>
</tr>
<tr>
<td>IDEA</td>
<td>228</td>
<td>24 (10.5%)</td>
<td>8</td>
<td>22</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Barnes-Hut</td>
<td>682</td>
<td>80 (11.7%)</td>
<td>25</td>
<td>123</td>
<td>38</td>
<td>0</td>
</tr>
<tr>
<td>CollisionTree</td>
<td>1032</td>
<td>233 (22.6%)</td>
<td>82</td>
<td>408</td>
<td>58</td>
<td>0</td>
</tr>
<tr>
<td>K-means</td>
<td>501</td>
<td>5 (1.0%)</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>5615</td>
<td>600 (10.7%)</td>
<td>143</td>
<td>898</td>
<td>269</td>
<td>2</td>
</tr>
</tbody>
</table>

8. Related Work

We group the related work into five broad categories: effect systems (not including ownership-based systems); ownership types (including ownership with effects); unique references; separation logic; and runtime checks.

Effect Systems: The seminal work on types and effects for concurrency is FX [33, 27], which adds a region-based type and effect system to a Scheme-like, implicitly parallel language. Leino et al. [30] and Greenhouse and Boyland [26] first added effects to an object-oriented language. None of these systems can represent arbitrarily nested structures or array partitioning, and they cannot specify arbitrarily large sets of regions. Also, the latter two systems rely on alias restrictions and/or supplementary alias analysis for soundness of effect, whereas DPJ does not.

Ownership Types: Some ownership-based type systems have been combined with effects to enable reasoning about noninterference. Both JOE [16, 46] and MOJU [14] have sophisticated effect systems that allow nested regions and effects. However, neither has the capabilities of DPJ's array partitioning and partially specified RPLs, which are crucial
to expressing the patterns addressed in this paper. JOE’s under effect shape is similar to DPJ’s +, but it cannot do the equivalent of our distinctions from the right. JOE allows slightly more precision than our rule LET when a type or effect uses a local variable that goes out of scope, but we have found that this precision is not necessary for expressing deterministic parallelism. MOJO has a wildcard region specifier ?, but it pertains to the orthogonal capability of multiple ownership, which allows objects to be placed in multiple regions. Leino’s system also has this capability, but without arbitrary nesting.

Lu and Potter [32] show how to use effect constraints to break the owner dominates rule in limited ways while still retaining meaningful guarantees. The any context of [32] is identical to Root:* in our system, but we can make more fine-grained distinctions. For example, we can conclude that a pair of references stored in variables of type C<R1: * > and C<R2: * > can never alias, if R1: * and R2: * are disjoint.

Several researchers [11, 3, 28] have described effect systems for enforcing a locking discipline in nondeterministic programs, to prevent data races and deadlocks. Because they have different goals, these effect systems are very different from ours, e.g., they cannot express arrays or nested effects.

Finally, an important difference between DPJ and most ownership systems is that we allow explicit region declarations, like [33, 30, 26], whereas ownership systems generally couple region creation with object creation. We have found many cases where a new region is needed but a new object is not, so the ownership paradigm becomes awkward. Supporting field granularity effects also is difficult with ownership.

Unique References: Boyland [13] shows how to use alias restrictions to guarantee determinism for a simple language with pointers. Terauchi and Aiken [48] have extended this work with a type inference algorithm that simplifies the type annotations and elegantly expresses some simple patterns of determinism. Alias restrictions are a well-known alternative to effect annotations for reasoning about heap access, and in some cases they can complement effect annotations [26, 12]. However, alias restrictions severely limit the expressivity of an object-oriented language. It is not clear whether the techniques in [13, 48] could be applied to a robust object-oriented language. Clarke and Wrigstad’s external uniqueness [17] is better suited to an object-oriented style, but it is not clear whether external uniqueness is useful for deterministic parallelism.

Separation Logic: Separation logic [40] (SL) is a potential alternative to effect systems for reasoning about shared resources. O’Hearn [35] and Gotsman et al. [25] use SL to check race freedom, though O’Hearn includes some simple proofs of noninterference. Parkinson [37] has extended C# with SL predicates to allow sound inference in the presence of inheritance. Raza et al. [39] show how to use separation logic together with shape analysis for automatic parallelization of a sequential program.

While SL is a promising approach, applying it to realistic programs poses two key issues. First, SL is a low-level specification language: it generally treats memory as a single array of words, on which notions of objects and linked data structures must be defined using SL predicates [40, 35]. Second, SL approaches generally enforce either require heavyweight theorem proving and/or a relatively heavy programmer annotation burden [37] or are fully automated, and thereby limited by what the compiler can infer [25, 39].

In contrast, we chose to start from the extensive prior work on regions and effects, which is more mature than SL for OO languages. As noted in [40], type systems and SL systems have many common goals but have developed largely in parallel; as future research it would be useful to understand better the relationship between the two.

Runtime Checks: A number of systems enforce some form of disciplined parallelism via runtime checks. Jade [43] and Prometheus [5] use runtime checks to guarantee deterministic parallelism for programs that do not fail their checks. Jade also supports a simple form of commutativity annotation [41]. Multiphase Shared Arrays [20] and PPL [47] are similar in that they rely on runtime checks that may fail if determinism is violated. None of these systems checks non-trivial sharing patterns at compile time.

Speculative parallelism [7, 23, 51] can achieve determinism with minimal programmer annotations, compared to DPJ. However, speculation generally either incurs significant software overheads or requires special hardware [38, 31, 50]. Grace [7] reduces the overhead of software-only speculation by running threads as separate processes and using commodity memory protection hardware to detect conflicts at page granularity. However, Grace does not efficiently support essential sharing patterns such as (1) fine-grain access distinctions (e.g., distinguishing different fields of an object, as in Barnes-Hut); (2) dynamically scheduled fine-grain tasks (e.g., ForkJoinTask); or (3) concurrent data structures, which are usually finely interleaved in memory. Further, unlike DPJ, a speculative solution does not document the parallelization strategy or show how the code must be rewritten to expose parallelism.

Kendo [36] and DMP [21] use runtime mechanisms to guarantee equivalence to some (arbitrary) serial interleaving of tasks; however, that interleaving is not necessarily obvious from the program text, as it is in DPJ. Further, Kendo’s guarantee fails if the program contains data races, and DMP requires special hardware support. SharC [6] uses a combination of static and dynamic checks to enforce race freedom, but not necessarily deterministic semantics, in C programs.

Finally, a determinism checker [44, 22] instruments code to detect determinism violations at runtime. This approach is not viable for production runs because of the slowdowns caused by the instrumentation, and it is limited by the cover-
age of the inputs used for the dynamic analysis. However, it is sound for the observed traces.

9. Conclusion

We have described a novel type and effect system, together with a language called DPJ that uses the system to enforce deterministic semantics. Our experience shows that the new type system features are useful for writing a range of programs, achieving moderate to excellent speedups on a 24-processor system with guaranteed determinism.

Our future goals are to exploit region and effect annotations for optimizing memory hierarchy performance; to add runtime support for more flexible operation on index-parameterized arrays; to add support for object-oriented parallel frameworks; and to add support for explicitly nondeterministic algorithms.

Acknowledgments

The following persons provided invaluable insight at various stages of this work: Ralph Johnson, Christopher Rodrigues, Marc Snir, Dan Grossman, Brad Chamberlain, John Brant, Rajesh Karmani, and Maurice Raibb.

References


A. Static Semantics Rules

We divide the static semantics in to five parts: rules for valid program elements (Figure 16), rules for validity, nesting, and inclusion of RPLs (Figure 17), rules for valid types and subtypes (Figure 18), rules for valid effects and subeffects (Figure 19), and rules for typing expressions (Figure 20).

B. Dynamic Semantics Rules

Figure 21 gives the rules for evaluating programs. If \( f : A \rightarrow B \) is a function, then \( f \subseteq \{ x \rightarrow y \} \) is the function \( f' : A \cup \{ x \rightarrow y \} \) defined by \( f'(a) = f(a) \) if \( a \neq x \) and \( f'(x) = y \). new(C) is the function taking each field of class \( C \) with type \( T \) to a null reference of type \( d\Gamma(T) \), and \( \text{new}(T[n]) \) is the function taking each index \( n' \in [0, n - 1] \) to a null reference of type \( d\Gamma(T)[i \leftarrow n'] \).

The rules for dynamic RPLs, types, and effects are nearly identical to their static counterparts. Instead of writing out all the rules, we describe how to generate them via simple substitution from the rules given in Section A. For every rule given there except RPL-REF, RPL-UNDERVAR, INCLUDE-PARAM, and INCLUDE-FULL, do the following: (1) append \( \text{DYNS} \) to the front of the name; (2) replace \( \Gamma \) with \( H \) and \( [i] \) with \( [n] \); and (3) replace \( R \) with \( dR \), \( T \) with \( dT \), and \( E \) with \( dE \). For example, here are the rules for dynamic class subtyping, generated by the substitution above from the rule \( \text{SUBTYPE-CLASS} \):

\[
\text{SUBTYPE-CLASS} : \frac{dR \subseteq dR'}{H \vdash \text{DYNS} \text{SUBTYPE-CLASS}}
\]

Then add the following rules:

\[
\text{DYNS-RPL-REF} : \frac{H \vdash o : dE}{H \vdash o : dE}
\]

\[
\text{DYNS-UNDERVAR} : \frac{H \vdash o : C<dR>}{H \vdash o : C<dR>}
\]

\[
\text{DYNS-TYPE-ARRAY} : \frac{H \vdash o \in [p] \subseteq dR}{H \vdash o \in [p] \subseteq dR}
\]

C. Soundness

C.1 Type and Effect Preservation

Definition 1 (Valid dynamic environments). A dynamic environment \( d\Gamma \) is valid with respect to heap \( H \) (\( H \vdash d\Gamma \)) if the following hold: (1) for every binding \( z \leftarrow o \in d\Gamma \), \( H \vdash o : dT \); (2) for every binding \( P \leftarrow dR \in d\Gamma \), \( H \vdash dR \); and (3) if \( \text{this} \leftarrow o \in d\Gamma \), then \( H \vdash o : C<dR> \) and \( \text{param}(C) \leftarrow dR \in d\Gamma \).

Definition 2 (Valid heaps). A heap \( H \) is valid if \( H \) for each \( o \in \text{Dom}(H) \), one of the following holds:

1. (a) \( H \vdash o : C<dR> \) and \( (b) H \vdash C<dR> \) and \( (c) \) for each field \( f \) in \( T \), \( f \in \text{def}(C) \), if \( H(o)(f) \) is defined, then \( H \vdash H(o)(f) : dT \) and \( H \vdash dT \) and \( H \vdash dT \leq T \langle o \leftarrow \text{this} \rangle<dR> \langle \text{param}(C) \rangle \rangle ; \) or

2. (a) \( H \vdash o : dT \langle \text{this} \rangle<dR> \langle \text{param}(C) \rangle \rangle ; \) or

Definition 3 (Instantiation of static environments). A dynamic environment \( d\Gamma \) instantiates a static environment \( \Gamma \) (\( \Gamma \vdash d\Gamma \leq \Gamma \)) if \( \Gamma \vdash \Gamma \), \( \Gamma \vdash H \), and \( \Gamma \vdash d\Gamma \); the same variables appear in \( \text{Dom}(\Gamma) \) as in \( \text{Dom}(d\Gamma) \); and for each pair \( z \leftarrow T \in \Gamma \) and \( z \leftarrow o \in d\Gamma \), \( H \vdash o : dT \) and \( H \vdash dT \leq d\Gamma(T) \).

Theorem 1 (Preservation). For a well-typed program, if \( \Gamma \vdash e : T \), \( E \) and \( H \vdash d\Gamma \leq T \), then \( \langle o, H', dE \rangle \), \( (a) H' \vdash o : dT \), \( (b) H' \vdash dT \leq d\Gamma(T) \), \( (c) H' \vdash dE \), and \( (d) H' \vdash dE \leq d\Gamma(E) \).
Figure 16. Rules for valid program elements. def(C, m) means the definition of method m in class C.

Figure 17. Rules for valid RPLs, nesting of RPLs, and inclusion of RPLs. The nesting and inclusion relations are reflexive and transitive (obvious rules omitted).

Figure 18. Rules for valid types and subtypes. def(C) means the definition of class C. T ≡ T' means that T and T' are identical up to the names of variables i.

Figure 19. Rules for valid effects and subeffects.

Figure 20. Rules for typing expressions. param(C) means the parameter of class C.
Figure 21. Rules for program evaluation.

Figure 22. Rules for disjointness of RPLs. The disjointness relation is symmetric (obvious rule omitted).

C.2 Disjointness

Figure 22 gives the rules for concluding that two static RPLs are disjoint; we extend them to dynamic RPLs as in Section B.

Definition 4 (Set interpretation of dynamic RPLs). Let \( b \) and \( H \vdash dR \). Then \( S(dR, H) \) is defined as follows: (1) \( S(dR_f, H) = \{ dR_f \} \); (2) \( S(dR : r, H) = \{ dR_f : r : dR_f \} \); (3) \( S(dR : [n], H) = \{ dR_f : [n] \} \); and (4) \( S(dR : *, H) = \{ dR_f : H \} \).

Definition 5 (Region of a field or array cell). If \( H \vdash o : C \langle dR \rangle \) and \( T_f \vdash n \in dR \in \text{def}(C) \), then \( \text{region}(o, f, H) = \{ dR_f : \text{this} = o \} \cap \text{param}(C) \cap dR_f \). If \( H \vdash o : dT \langle dR \rangle \), then \( \text{region}(o, n, H) = dR_i(n \leftarrow n) \).

Proposition 1 (Disjointness of region sets). If \( H \vdash dR \neq dR' \), then \( S(dR, H) \cap S(dR', H) = \emptyset \).

Proposition 2 (Distinctness of disjoint regions). If \( H \vdash \text{region}(o, f, H) \neq \text{region}(o', f', H) \), then either \( o \neq o' \) or \( f \neq f' \). And if \( H \vdash \text{region}(o, n, H) \neq \text{region}(o', n', H) \), then either \( o \neq o' \) or \( n \neq n' \).

C.3 Noninterference of Effect

Figure 23 gives the noninterference relation on static effects. We extend this relation to dynamic effects as in Section B.

Theorem 2 (Soundness of noninterference). If \( \Gamma \vdash e : T, E \) and \( \Gamma \vdash e' : T', E' \) and \( \Gamma \vdash E \neq E' \) and \( H \vdash d\Gamma \leq \Gamma \) and \( (e, d\Gamma, H) \rightarrow (o, H', dE) \) and \( (e', d\Gamma', H') \rightarrow (o', H'', dE') \), then there exists \( H'' \) such that \( (e', d\Gamma', H'') \rightarrow (o', H'', dE') \) and \( (e, d\Gamma, H'') \rightarrow (o, H'', dE) \).
CRUISE: Cache Replacement and Utility-aware Scheduling

Aamer Jaleel†, Hashem H. Najaf-abadi*, Samantika Subramaniam†, Simon C. Steely Jr.,† and Joel Emer†‡

†Intel Corporation, VSSAD
Hudson, MA
‡Intel Corporation, VPG
Folsom, CA
‡Massachusetts Institute of Technology (MIT)
Cambridge, MA

{aamer.jaleel, hashem.hajemi}, samantika.subramaniam, simon.c.steely.jr, joel.emer)@intel.com

Abstract

When several applications are co-scheduled to run on a system with multiple shared LLCs, there is opportunity to improve system performance. This opportunity can be exploited by the hardware, software, or a combination of both hardware and software. The software, i.e., an operating system or hypervisor, can improve system performance by co-scheduling jobs on LLCs to minimize shared cache contention. The hardware can improve system throughput through better replacement policies by allocating more cache resources to applications that benefit from the cache and less to those applications that do not.

This study presents a detailed analysis on the interactions between intelligent scheduling and smart cache replacement policies. We find that smart cache replacement reduces the burden on software to provide intelligent scheduling decisions. However, under smart cache replacement, there is still room to improve performance from better application co-scheduling. We find that co-scheduling decisions are a function of the underlying LLC replacement policy. We propose Cache Replacement and Utility-aware Scheduling (CRUISE)—a hardware/software co-designed approach for shared cache management. For 4-core and 8-core CMPs, we find that CRUISE approaches the performance of an ideal job co-scheduling policy under different LLC replacement policies.

Categories and Subject Descriptors D.4.1 [Process Management]: Scheduling, B.3.2 [Design Styles]: Cache memories, C.1.4 [Parallel architectures]: Distributed architectures

General Terms Algorithms, Measurement, Performance, Design.

Keywords Scheduling, Cache Replacement, Shared Cache

1. Introduction

Emerging technologies such as virtualization, multi-core, and multi-socket systems have enabled the consolidation of multiple applications onto a single system. In doing so, a wide variety of applications with differing memory demands can concurrently execute and compete for shared resources in the system. Since modern multi-core and multi-socket CMP systems typically contain one or more shared last-level caches (LLC), system performance is typically determined by how well the shared LLC is managed.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

© 2012 ACM 978-1-4503-0759-8/12/03. ..$10.00.

Traditionally, there have been two approaches to manage shared LLCs: intelligent software scheduling [37, 38, 31, 32, 7, 15, 17, 18, 21] and smart hardware cache replacement [4, 12, 10, 11, 14, 25, 34, 36]. Software scheduling puts the onus of co-scheduling applications on a layer between the hardware and the applications themselves, like the operating system or a hypervisor. The advantage of software scheduling is that the underlying hardware is agnostic to the scheduling of applications and can be designed independently. However, the drawback is that the scheduler is unaware of the runtime memory requirements of the applications scheduled. To address this problem, researchers have proposed using existing hardware performance counters (or adding new counters) to guide scheduling [31, 21, 38]. For example, a recent proposal measures per-application miss-rate using performance counters and then co-schedules applications based on the degree of memory intensity [38]. However, this approach can cause sub-optimal co-scheduling decisions since per-application miss-rate while sharing a cache can be significantly different from the miss-rate while running in isolation. Nonetheless, hardware performance counter based scheduling algorithms, to some degree, have addressed shared cache contention.

Researchers from the hardware community, on the other hand, have addressed heterogeneity in memory demand by proposing to build resource sharing decisions directly into the cache replacement policy. Such smart replacement policies dynamically allocate more cache resources to applications that benefit from the cache and less to applications that do not benefit from the cache. While cache replacement does not specify which applications can be co-scheduled to share a cache resource, it can change the impact that these applications have on shared cache usage. Consequently, we believe that it is important to study the interaction between the software and hardware approaches to shared cache management.

This paper presents the first comprehensive study on the interactions between intelligent software scheduling and smart cache replacement. In general, we find that smart cache replacement policies minimize the burden on software to provide intelligent scheduling decisions. However, they do not eliminate the need for intelligent scheduling decisions. Based on our observations, we propose a hardware/software co-designed approach for shared cache management called Cache Replacement and Utility-aware Scheduling (CRUISE). We also propose a novel hardware mechanism, Runtime Isolated Cache Estimator (RICE), which provides a low overhead mechanism to dynamically estimate the isolated cache performance of an application while still sharing the cache with other applications. RICE requires no changes to the existing shared cache structure and merely requires two counters per application. The software component of our proposal uses cache utility information provided by RICE and the knowledge of the underlying LLC replacement policy to intelligently co-schedule
Figure 1: 4-core CMP with Two Shared LLCs.
applications. Our evaluations on 4-core and 8-core CMPs with two
and four threads sharing an LLC show that CRUISE consistently
provides near-optimal co-scheduling decisions.

2. Motivation
Recent studies have shown that the commonly used LRU
replacement policy (and its approximations) perform poorly on
shared caches [12, 34, 10, 30, 29, 25]. This is because LRU
replacement allocates cache resources based on demand instead of
cache utility [30, 25, 12]. To improve shared cache performance,
researchers from the computer architecture community propose smart
replacement policies to improve upon LRU [12, 34, 10, 30, 29, 25].
On the other hand, other researchers have proposed intelligent co-
scheduling policies [37, 38, 32, 31, 21] that an operating system or
hypervisor can use to minimize shared cache contention.

Figure 2 illustrates a 4-core CMP with two shared LLCs. When
four applications (A, B, C, and D) concurrently execute on this CMP,
there are three possible application schedules: AB/CD, AC/BD, and
AD/BC. Depending on the amount of shared cache contention, the
three application schedules may observe different system and per-
application performance. For a given performance metric (e.g.,
throughput, weighted speedup), we refer to the application schedule
that yields the best system performance as Optimal Application Schedule (OAS) and the application schedule that yields the lowest
system performance as Worst Application Schedule (WAS).

In our 4-core CMP system, application performance and system
performance is largely dependent on the degree of cache contention
between the two applications co-scheduled (i.e. share) on the same
LLC. To avoid the worst application schedule, conventional wisdom
prevents co-scheduling a CPU bound application with a memory
bound application. However, such practice is only necessary for an
LRU managed LLC. In an LLC managed by smart cache
replacement, co-scheduling a CPU bound application with a memory
bound application no longer hurts the performance of a CPU bound
application. This is because smart cache replacement policies allocate
cache resources based on utility instead of demand. Since the
majority of existing intelligent scheduling studies have been
evaluated in the presence of an inefficient LLC replacement policy
(i.e., LRU), the question arises as to whether intelligent application
scheduling is necessary in the presence of smart cache replacement.

In efforts to understand the interaction between cache
replacement and application scheduling, Figure 2 illustrates the
effects of optimal scheduling in the presence and absence of smart
cache replacement. The study is based on 1565 4-core heterogeneous
SPEC CPU2006 application mixes simulated on our 4-core CMP
illustrated in Figure 1. We use DRRIP [10], a recently proposed low
overhead, high performing shared cache replacement policy.

In the figure, the x-axis represents the system performance ratio
OAS_{LRU}/WAS_{LRU} while the y-axis represents the system
performance ratio OAS_{DRRIP}/WAS_{DRRIP}. This ratio (illustrated as a
percent) represents the performance variability that intelligent
scheduling policies attempt to minimize. We use system throughput
as our performance metric. Each data point in the figure represents a
4-core workload mix. Some mixes are emphasized using large shapes
(circle, square, diamond, and triangle) to highlight representative
behavior. Circles are in “Region I”, squares are in “Region II”,
diamonds in “Region III”, and finally triangle in “Region IV”.

An interesting observation from the figure is the behavior of
workloads in Region II (emphasized by squares). Under LRU
replacement, these workloads have significant performance variation
(up to 28% on the y-axis). Hence, under LRU, these workloads
significantly benefit from intelligent scheduling decisions. However,
under a smarter replacement policy, like DRRIP, these workloads
have lesser performance variation (<4% on the y-axis).
Consequently, under DRRIP, there is little benefit from improving
scheduling decisions. Upon inspection, these workload mixes consist
of memory bound applications paired with CPU bound applications.
Smart cache replacement policies, such as DRRIP, are particularly
designed to address such workload mixes.

In general, the figure shows that the majority of workload mixes
lie below the linear bisector. Such behavior suggests that smarter
cache replacement policies minimize the burden on software (i.e.
operating system or hypervisor) to provide intelligent co-scheduling
decisions. However, note that smarter replacement policies do not
eliminate the need for intelligent scheduling decisions. For example,
workloads in Region IV (emphasized by triangle) can have as much
as 20% performance variation under both LRU and DRRIP.
Furthermore, for workload mixes in region III, smart replacement
policies introduce up to 20% performance variation. Note that these
workload mixes had less than 4% performance variation under LRU
replacement. Upon inspection, these workload mixes consist of
several non-memory bound applications that benefit from more
intelligent co-scheduling decisions.

For the workloads studied, we find that DRRIP alone improves
performance over LRU by roughly 4.8%. Intelligent scheduling by
itself in an LRU-managed cache improves performance by 4.4%. We
find there is up to 8.4% performance potential when intelligent
scheduling is employed in a DRRIP-managed cache. This shows that

Figure 2: Behavior of Optimal Co-Scheduling in Presence and
Absence of Intelligent Replacement. Some workload mixes are
emphasized with different sized shapes and are analyzed more
closely in the results section of this paper.
the underlying replacement policy and intelligent scheduling combined can together improve performance significantly. The next section proposes such an intelligent scheduling policy.

3. Cache Replacement and Utility-aware Scheduling (CRUISE)

Determining a suitable application co-schedule requires runtime knowledge on the cache requirements of an application in isolation [31, 38]. The next section describes our run-time mechanism to determine isolated cache utility of an application. Given the cache utility information, we then present our Cache Replacement and Utility-aware Scheduling (CRUISE) proposal that co-schedules applications based on knowledge of the underlying LLC replacement policy. Finally, we discuss how an operating system (or hypervisor) can incorporate CRUISE to improve system performance.

3.1. Classifying Application Cache Utility

Recent work used colors [18] and animals [35] as analogies to characterize the memory intensity of applications. Instead of using analogies, we propose to classify applications based on their cache performance on the available CMP cache hierarchy. We categorize application cache utility into four different categories:

- **“Core Cache Fitting” (CCF) Applications**: These applications have a working set size that fits in the smaller levels of the cache hierarchy. CCF applications have no benefit from the shared LLC.
- **“LLC Thrashing” (LLCT) Applications**: These applications have a working set size that is greater than the available LLC. We consider streaming applications as LLCT applications. Under PPC, LLCT applications degrade performance of any application that benefits from the shared LLC.
- **“LLC Fitting” (LLFC) Applications**: These applications require the majority of the available shared LLC capacity to perform well. If these applications do not receive the bulk of the shared LLC, like LLCT applications, their performance degrades significantly due to cache thrashing. As such, LLFC applications perform best when running in isolation or when sharing the LLC with CCF applications. LLCF application performance degrades when co-executed with any other application.
- **“LLC Friendly” (LLCF) Applications**: These applications benefit from the available shared LLC and continue to do so as they are given more cache resources. Unlike LLFC applications, LLCF application performance does not degrade significantly when they do not receive the bulk of the shared LLC. LLCF application performance degrades only when co-executed with LLC or LLCF applications.

Based on the above application classifications, intelligent scheduling decisions can be developed to manage a system with multiple shared LLCs. For example, since CCF applications do not require the shared LLC, co-scheduling them with LLCF applications can significantly improve LLCF application performance. Similarly, it is best to co-schedule LLCF applications with LLCF applications.

Before we provide a detailed description of our proposed application co-scheduling policies, we first discuss mechanisms to classify application cache utility.

3.1.1 Profiling based Classification of Applications

Cache utility of an application can be determined statically using profile information. Applications can be executed beforehand and classified into one of the four categories described above. However, the primary drawback of a profiling based approach is that the information gathered is highly sensitive to the choice of input sets (which is only available at run time), application phase, and also varies across different types of applications.

3.1.2 Runtime Classification of Applications

A straightforward way of classifying application cache utility in isolation is to periodically pause all cores on a CMP and measure the application cache performance. However, this approach degrades performance of all other concurrently executing applications. Alternatively, external shadow tags can be used to monitor cache utility [25, 35]. However, such approaches require extra hardware and power overhead for the shadow tags. To avoid additional overhead and changes to the existing shared cache structure, we propose a Runtime Isolated Cache Estimator (RIC). RICE estimates cache utility of an application by using a Set Dueling Monitor (SDM) [24]. An SDM estimates the misses for any given policy by permanently dedicating a few sets1 of the cache to follow that policy. RICE dedicates two SDM per application in the cache. For each SDM, two 32-bit counters track the number of accesses and misses to that SDM (illustrated in Figure 3). The first SDM, referred to as Full-SDM (FSDM) only allows A to allocate lines in these cache sets. All other applications are required to bypass these cache sets. The second SDM, referred to as Half-SDM (HSDM) estimates the caching behavior of an application if it were to have sole access to the cache. The FSDM for an application A (FSDM^A) only allows A to allocate lines in these cache sets. All other applications are required to bypass these cache sets. The second SDM, referred to as Half-SDM (HSDM) estimates the caching behavior of an application if it were to have sole access to only half the cache. All other applications can share the remaining ways in the HSDM. An HSDM can be implemented using way partitioning [25]. For example, in FSDM^A of a 4-way set associative cache, application A only allocates lines in way 0 and way 1 while all other applications allocate lines in way 2 and way 3.

\[ \text{classfun = LCCF, if (FSDM^A_{MR} < \alpha) \text{classfun = CCF}} \]
\[ \text{else if (FSDM^A_{APR} \geq \beta)} \{ \text{classfun = LLCT}\]
\[ \text{if (FSDM^A_{MR} < \epsilon)} \{ \text{classfun = LCCFR}\]
\[ \text{if (HSDM^A_{MR} > \gamma) \text{classfun = LCCF}} \]

Figure 3: Runtime Isolated Cache Estimator (RICE) and Classification Algorithm. In the classification algorithm, to compute APKI, we multiply FSDM^A ACC by (total sets / SDM size). For this paper, we use \( \alpha = 1, \beta = 4, \epsilon = 25, \gamma = 50.\)

---

1. Prior work has shown that 32 sets are sufficient to estimate cache performance [25, 24]. Throughout the paper an SDM consists of 32 sets.

2. For an inclusive LLC, a single way in each set can be dedicated for all other applications. This effectively causes the SDM to behave like a direct mapped cache for all other applications.
RICE can be enabled permanently or can be software controlled through privileged instructions. Providing the dynamic on/off ability reduces RICE overhead especially when the FSDM or HSDM tend to be “hot” sets for other applications. The drawback however is that privileged instructions must be introduced into the instruction set architecture (ISA) to explicitly enable or disable RICE.

When RICE is enabled, periodically sampling the RICE counters and calculating the accesses per kilo instructions (APKI) and miss rate (MR) metrics in each SDM can be used to classify the cache utility of application A at runtime. For example, if the FSDM<sub>APKI</sub> application is small (<i>e.g.</i> < 1), then the application is a CCF application. On the other hand, if the FSDM<sub>APKI</sub> application is high, the FSDM<sub>MR</sub> is small but the HSDM<sub>MR</sub> miss rate is high, then the application is a LLCF application. However, if the FSDM<sub>APKI</sub> application is high and the FSDM<sub>MR</sub> is also very high, then the application is an LLCT application. Even none of the above conditions apply, then the application can be classified as a LLCF application. Note that the primary use of the HSDM is to classify an LLCF application. Without the HSDM, an LLCF application would incorrectly be classified as an LLCF application. Since LLCF applications behave like LLCT applications in the presence of any other application, it is imperative that such applications be classified correctly. A summary of our runtime application classification algorithm is provided in Figure 3.

3.2. Cache Replacement and Utility-aware Scheduling

Given statically profiled or dynamically gathered cache utility information of all concurrently executing applications, we now discuss the design of an intelligent scheduling algorithm that is co-designed to the underlying shared LLC replacement policy.

3.2.1 CRUISE for Demand-based Cache Replacement

LRU-managed shared caches allocate cache resources based on demand instead of cache utility. As a result, under LRU replacement, LLCT applications receive more cache resources despite the fact that they do not benefit from the cache. Since LLCT applications can degrade the performance of LLCF and LLCR applications, it is best to co-schedule them with LLCT or CCF applications.

When scheduling CCF applications, it is best to separate these applications across all available shared LLCs. Since CCF applications require little LLC resources, such a strategy allows co-scheduled applications to utilize more of the LLC.

When scheduling LLCF applications, it is best to co-schedule them with CCF applications. This is because an LLCF application requires majority of the shared cache while a CCF application requires little LLC resources. In the absence of CCF applications, LLCF applications behave like LLCT applications and thus should be scheduled in a similar fashion as LLCT applications.

Finally, LLCF applications perform well when co-scheduled with other applications that benefit from the cache. Once LLCT, CCF, and LLCF applications are scheduled onto the appropriate LLCs, LLCF applications can be co-scheduled anywhere.

Since an LRU-managed shared LLC allocates cache resources on demand, we propose CRUISE-LRU (CRUISE-L). To ensure a proper schedule, steps must be followed in the order listed below:

1. Group LLCT applications on the same LLC
2. Spread CCF applications across all LLCs
3. Co-schedule LLCF with CCF applications
4. Fill in the LLCF applications

During the scheduling process, if all computing resources on one shared LLC are occupied, then the scheduling algorithm overflows onto the next available LLC (selected at random). Additionally, if there exists more than one application from the same category, then the scheduling algorithm selects an application at random.

3.2.2 CRUISE for Utility-based Cache Replacement

Unlike LRU, DRRIP-managed shared caches allocate cache resources based on utility instead of demand. As a result, LLCT applications receive very few cache resources when co-scheduled with applications that benefit from the shared cache. Consequently, in a DRRIP-managed cache, LLCF applications can be treated like CCF applications that do not benefit from the available cache.

When co-designing the scheduling policy for an LRU-managed cache, an entire shared cache was used to “contain” LLCT applications and prevent them from hurting LLCF and LLCT applications. In doing so, CRUISE-L is inefficient since it can “waste” one or more shared caches. However, in a system with more than one DRRIP-managed shared cache, we propose CRUISE-DRRIP (CRUISE-D) to efficiently utilize all available shared caches. Again, to ensure a proper schedule, steps must be followed in the order listed below:

1. Spread LLCT applications across all LLCs
2. Spread CCF applications across all LLCs
3. Co-schedule LLCF with CCF/LLCT applications
4. Fill in the LLCF applications

Note that CRUISE-D is similar to the recently proposed Distributed Intensity (DI) proposal [38]. Like DI, CRUISE-D separates LLC applications onto separate LLCs. However, CRUISE-D differs from DI in that it explicitly detects LLCF applications and co-schedules them intelligently to maximize system performance.

3.2.3 CRUISE-L vs. CRUISE-D

CRUISE-L and CRUISE-D primary differ in how they co-schedule LLCT applications. CRUISE-D treats LLCT applications like CCF applications. This is because a DRRIP managed cache naturally allocates fewer cache resources to LLCT applications since they do not benefit from cache.

Note that while CRUISE-L is co-designed for an LRU-managed cache, CRUISE-L is applicable for any unmanaged cache replacement policy. Similarly, though CRUISE-D is co-designed for a DRRIP-managed cache, CRUISE-D is applicable for any utility-based cache replacement policy. Therefore, applying CRUISE-L to a utility-managed cache or applying CRUISE-D to an unmanaged cache will yield suboptimal co-schedules.

3.3. Integrating CRUISE into Existing Software

Existing software (i.e., operating systems or hypervisor) can utilize CRUISE by using statically profiled utility information or by dynamically learning the application utility if RICE hardware is available. If an operating system implements CRUISE using statically profiled cache utility information, then the natural opportunity to determine the best application co-schedule is when the operating system (or hypervisor) schedules a new application onto a CPU (e.g., after a context switch). With statically profiled cache utility information, co-scheduling decisions are limited to context switches and/or when an application finishes execution. As a result, the operating system (or hypervisor) cannot adapt to dynamic application phases.
Alternatively, if the underlying processor supports RICE, then an operating system (or hypervisor) can dynamically adapt to application phase behavior. To do so, the operating system can periodically read and classify all applications using the RICE counters. Upon classifying application cache utility, the operating system (or hypervisor) can periodically apply CRUISE.

4. Experimental Methodology

We use CMPsim [9], a Pin [19] based trace-driven x86 simulator for our performance studies. Our baseline system is a 4-core CMP with two shared LLCs (see Figure 1). Each core in the CMP is a 4-way out-of-order processor with a 128-entry reorder buffer and a three level cache hierarchy. We assume single-threaded cores with the L1 and L2 caches private to each core. The L1 instruction and data caches are 4-way 32KB each while the L2 cache is unified 8-way 256KB. The L1 and L2 cache sizes are kept constant in our study. We support two L1 read ports and one L1 write port on the data cache. Each last-level cache (LLC) is a unified 16-way 4MB cache that is shared by two cores in the CMP. We assume a banked LLC with as many banks as there are cores in the system. All caches in the hierarchy are non-inclusive and use a 64B line size. For replacement decisions, the L1 and L2 caches always use the Not Recently Used3 (NRU) replacement policy. To evaluate interactions with intelligent scheduling, we only vary the LLC replacement policy between NRU and DRRIP [10]. We model a stream prefetcher that trains on L2 cache misses and prefetches lines into the L2 cache (i.e., the prefetcher is private to each core). The prefetcher has 16 stream detectors. The load-to-use latencies for the L1, L2, and LLC are 1, 10, and 24 cycles respectively. We model an interconnect with a fixed average latency. Bandwidth onto the interconnect is modeled using a fixed number of MSHRs. Contention for the MSHRs models the increase in latency due to additional traffic introduced into the system. We use a queuing model to model off-chip contention. We model a 150 cycle unaligned latency penalty to main memory and support 16 outstanding misses to memory. The cache hierarchy organization and latencies are based on the Intel Core i7 processor [2]. Note that our proposed scheduling policies do not rely on the specific latencies used.

We do not use an operating system, but instead augment our CMPsim model with an application scheduler. Applications are initially assigned to the first available free CPU at the beginning of simulation. Our dynamic scheduler periodically (every 1 ms) determines whether applications need to be re-scheduled. When classifying an application at runtime, RICE uses hysteresis to ensure steady state. This is to prevent CRUISE from doing frequent reschedulings due to slight phase changes in the application. When an application is re-scheduled to a new CPU, we model all compulsory misses to warm up all levels of the cache hierarchy on the new CPU (and possibly new LLC). Additionally, when CRUISE attempts to reschedule application, CRUISE tries to minimize the number of applications that need to be rescheduled. This is to minimize the compulsory miss overhead for all applications.

4.1. Benchmarks

For our study, we use benchmarks from the SPEC CPU2006 suite. We first grouped the SPEC CPU2006 benchmarks into four different categories based on their L1, L2, and LLC cache hit behavior. Of all the SPEC CPU2006 benchmarks, we selected few applications from each category to cover the spectrum of hit/miss behavior in the different levels of the cache hierarchy. A total of 15 representative SPEC CPU2006 benchmarks were selected. Each benchmark was compiled using the icc compiler with full optimization flags. Representative regions for the SPEC benchmarks were all collected using PinPoints [23]. Table II lists the 15 SPEC CPU2006 benchmarks and their misses per 1000 instructions (MPKI) in the L1, L2, and LLC when run in isolation. To illustrate application cache utility, the MPKI numbers are reported in the absence of a prefetcher.

To evaluate our proposed scheduling algorithms, we ran all possible four-threaded combinations of the 15 SPEC CPU2006 benchmarks, i.e. 15 choose 4—1356 workloads. To provide insights on when scheduling policies are beneficial, we selected 26 workload mixes (listed in Table I) to showcase results. These 26 workload mixes correspond to the same workload mixes emphasized with different shapes in Regions I, II, III, and IV of Figure 2. Recall that Region I workload mixes have no performance variation under different application schedules. Region II workload mixes have significant performance variation under LRU-managed shared caches. These workload mixes mostly consist of LLC applications. Region III workload mixes have significant performance variation under DRRIP-managed shared caches. These workload mixes mostly consist of LLCFR applications. Finally Region IV workload mixes

---

3. NRU is the hardware approximation for LRU replacement. NRU performs similar to LRU for a wide variety of workloads [10] and is the commonly used LLC replacement policy in majority of microprocessors today [10].
have performance variation under both replacement policies. These workload mixes consist of both LLCT and LLCFR applications. To provide a thorough analysis, we also provide results for all 1365 workloads wherever applicable.

We simulated half billion instructions for each benchmark. Simulations continue to execute until all benchmarks in the workload mix execute at least half billion instructions. If a faster thread finishes its required instructions, it continues to execute to complete for cache resources. We only collect statistics for the half billion instructions committed by each application. This methodology is similar to existing work on shared cache management [25, 12, 34].

4.2. Metrics

For our studies we use the three metrics commonly used in literature for measuring the performance of multiple concurrently executing applications: throughput, weighted speedup, and fairness. The "harmonic mean fairness" metric (which is harmonic mean of normalized IPCs) balances both fairness and performance [20]. The different metrics are defined as follows:

\[
\text{Throughput} = \Sigma \text{IPC}_i \quad \text{(Eq 1)}
\]

\[
\text{Weighted Speedup} = \Sigma (\text{IPC}_i / \text{SingleIPC}_i) \quad \text{(Eq 2)}
\]

\[
\text{Harmonic Mean Fairness} = N / \Sigma (\text{SingleIPC}/\text{IPC}_i) \quad \text{(Eq 3)}
\]

where \(\text{IPC}_i\) is the IPC of the \(i\)th application when it concurrently executes with other applications and \(\text{SingleIPC}_i\) is IPC of the same application in isolation.

5. Results and Analysis

5.1. Throughput

To decouple the performance of CRUISE and the accuracy of our application classifier RICE, we first illustrate CRUISE using cache utility information from static profiling. In doing so, we can measure the overall accuracy of the CRUISE algorithm that is specifically co-designed to the replacement policy. RICE is just our low overhead hardware classification mechanism.

Figure 4 presents the throughput of five different application scheduling policies (random, CRUISE-L, CRUISE-D, Distributed Intensity (DI)\(^4\) [38], and Optimal Application Schedule (OAS)) normalized to the Worst Application Schedule (WAS). Random schedule refers to selecting a co-schedule at random from the different possible co-schedules. The x-axis represents the different workload mixes. The bar labeled geometric is the geometric mean of all 26 workloads. In an LRU-managed LLC (Figure 4a), as expected, workload mixes from category I do not benefit from scheduling since all applications in these workload mixes do not benefit from the shared LLC. Workload mixes from category III also do not benefit from scheduling because they mostly consist of LLC applications. For such applications, LRU works best. Workload mixes from categories II and IV benefit from intelligent scheduling decisions because they consist of LLCT applications. In these workloads, we find that CRUISE-L performs very similar to OAS. In some scenarios CRUISE-L does not perform as well as OAS (e.g., MIX_04, MIX_06, MIX_19). CRUISE-L reaches suboptimal decision because these workload mixes consist of multiple applications that belong to the same cache utility category (e.g.,

\footnote{4 For DI, we use isolated cache miss rate based on profile information.}

Figure 4: Throughput Comparison of Scheduling Policies to Worst Application Schedule (WAS) on a 4-core CMP. (a) LRU-managed LLC (b) DRRIP-managed LLC
multiple LLCPR applications). When co-scheduling applications, CRUISE randomly selects an application and does not distinguish between applications that belong to the same cache utility category. Though further enhancements can be made to CRUISE to make this distinction, the first order benefits are from distinguishing applications that belong to different cache utility categories.

Additionally, the figure also shows that DI scheduling does not perform as well as CRUISE-L (e.g. MIX_05, MIX_07, MIX_20). Unlike CRUISE-L, CRUISE-D is specifically designed for a utility managed cache and thus does not perform well on a demand managed cache.

The figure also shows that DI scheduling does not perform well as well as CRUISE-L (e.g. MIX_05, MIX_07, MIX_09). This is because these workload mixes consist of an LLCPR application, and DI incorrectly co-schedules LLCPR applications. For example, consider MIX_05, the workload mix consists of applications lib, sje, spb, wrf. Based on cache miss rates, DI co-schedules these applications as lib, sje, spb, wrf. However, the OAS is spb, sje, lib, wrf. This is because spb is an LLCPR application that benefits from being co-scheduled with sje, a CCP application. Furthermore, DI has a suboptimal performance under LRU because it co-schedules memory bound applications with CPU bound applications. In doing so, DI degrades CPU bound application performance. As a result, DI is unable to arrive at the OAS.

On average in an LRU-managed cache, DI performs similar to random scheduling, both performing roughly 4% better than WAS. CRUISE-D and CRUISE-L perform roughly 3% and 8% better than WAS respectively. CRUISE-L bridges 90% of the gap between WAS and OAS — OAS performs roughly 9% better than WAS.

Figure 4b shows the performance of the different scheduling policies on a DRRIP-managed cache. Recall that DRRIP allocates cache resources based on utility instead of demand. We again observe that workloads from category 1 do not benefit from scheduling since these mix runs do not benefit from the shared LLC. Unlike an LRU-managed cache, workload mixes in category II do not benefit from scheduling because DRRIP inherently knows how to handle LLCPR applications in a workload mix. However, CRUISE-D finds opportunity to improve workload mixes from categories III and IV because they consist of multiple LLCPR and LLCCT applications. In a DRRIP-managed cache, we again observe CRUISE-D almost always outperforms CRUISE-L. In a utility managed shared LLC, random, DI, CRUISE-L, and CRUISE-D improve performance over WAS by 4.4%, 3.7%, 3.8%, and 7.5% respectively. Again, note that CRUISE-

D bridges 90% of the performance gap between WAS and OAS — OAS performs roughly 8% better than WAS.

To illustrate the behavior of CRUISE-L and CRUISE-D across a much wider set of workload mixes, Figure 5 shows per-workload performance comparison of each scheduling policy on an LRU-managed and DRRIP-managed cache. The x-axis in the figures show the 1365 workload mixes and the y-axis shows the performance relative to OAS. Values at 1 imply that CRUISE application schedule is identical to OAS, and values below 1 illustrate the relative throughput difference between CRUISE and OAS. Each graph is sorted in ascending order based on the CRUISE algorithm. Both figures show that CRUISE significantly reduce the performance variation between WAS and OAS. For workload mixes where the performance variation is significantly high, CRUISE frequently arrives at an optimal schedule. In both LRU-managed and DRRIP-managed shared caches, the majority of performance variation is less than 5%. For cases where the performance variation is greater than 5%, further refinement can be made by intelligently selecting between applications that belong to the same cache utility category.

5.2. Weighted Speedup and Fairness

Again, assuming statically profiled utility information, Figure 6 and 7 presents the throughput, weighted speedup, and fairness comparison of CRUISE on both LRU-managed and DRRIP-managed caches. Each figure shows the workloads on the x-axis and the corresponding performance metric on the y-axis. Like Figure 5, the throughput and weighted speedup metrics in these figures are normalized to OAS. The throughput and weighted speedup figures compare five different scheduling policies: WAS, random, CRUISE-L, CRUISE-D, and DI. We present 's-curves' independently sorted for each scheduling algorithm.

In an LRU-managed cache, across all 1365 workloads, for the throughput and weighted speedup metrics, there is roughly 0.5%
performance variation for CRUISE-L, roughly 2.5% for CRUISE-D, random, and DI scheduling, and finally roughly 4% for WAS. Similarly, for a DRRIP-managed cache, we observe roughly 0.8% performance variation for CRUISE-L, roughly 1.8% for CRUISE-D, random, and DI scheduling, and finally roughly 3.5% for WAS. Also, note that a randomly selected co-schedule tends to always be better than the WAS. However, CRUISE-D and CRUISE-L always provide better performance than a random schedule. Additionally, both CRUISE-L and CRUISE-D are within 1% of OAS for the fairness metric. In summary, Figure 6 and Figure 7 reveal that CRUISE performs similar to OAS for the throughput, weighted speedup, and fairness metrics—essentially CRUISE is robust across all metrics.

5.3. Scalability

We now discuss scalability of CRUISE by varying the number of shared LLCs and also varying the number of threads sharing an LLC. We construct 8-core workload mixes and consider two 8-core systems (a) four shared LLCs—two cores each sharing an LLC (b) two shared LLCs—four cores each sharing an LLC. For both these systems, we evaluate the performance of different scheduling policies for LRU and DRRIP-managed caches. We evaluate all possible 8-core combinations of the 15 applications—6435 workload mixes. For each system, we also evaluate all possible co-schedules. For a four LLC system with two cores per LLC, there are 105 possible co-schedules while for a two LLC system with eight cores per LLC, there are 35 possible co-schedules. Since we have already illustrated that CRUISE performs well across all performance metrics, we now limit ourselves to the throughput metric. We observe similar behavior for both the weighted speedup and fairness metrics.

Figure 8 illustrates CRUISE performance for the different 8-core configurations. As the number of shared LLCs increase, CRUISE consistently bridges the performance variation between OAS and WAS (Figure 8 (a) and (b)). Similarly, when increasing the number of threads per LLC, CRUISE still manages to bridge the performance variation (Figure 8 (c) and (d)). In these larger systems, CRUISE consistently performs better than all other scheduling policies.

5.4. Dynamic Classification

Thus far we have investigated the accuracy of CRUISE using profiled utility information. We now evaluate CRUISE with dynamically observed utility information gathered by RICE. Note that any performance deviation from OAS can primarily be attributed to the accuracy and sampling overhead of RICE. Figure 9 illustrates CRUISE-L and DI performance compared to OAS for our baseline 4-core CMP with two shared LLCs. We do not account for the RICE overhead due to sampling for OAS. We observe that CRUISE-L consistently tracks the performance of OAS, thereby showing that RICE performs well in dynamically classifying the application. We also observe that CRUISE-L consistently outperforms DI. On average, we find that CRUISE-L, DI, and OAS improve performance over WAS by roughly 2%, 0.2%, and 4% respectively. The difference between CRUISE-L and OAS is roughly 2% due to the sampling overheads of RICE (we observed similar comparison of CRUISE-D to OAS in DRRIP-managed caches). Note that these single digit performance gains are not insignificant since our studies span more than 1000 workload mixes that include a significant number of workloads that do not benefit from intelligent scheduling.

Figure 8: Performance of CRUISE when Increasing Number of Shared LLCs or when Increasing Number of Cores Per LLC (C/LLC). Note that these curves are independently sorted for each scheduling policy.
Both CRUISE-L and DI degrade performance slightly (less than 5%) compared to WAS. The negative outliers can be attributed to the sampling overhead of RICE (for CRUISE-L) or the overhead of compulsory misses from to context switches. Our investigations revealed that for the duration of the runs, the application classification reaches steady state and the number of context switches are low. This indicates that there is opportunity to reduce the negative overhead and improve CRUISE performance by periodically sampling RICE counters. Investigating this is part of ongoing work.

5.5. Sensitivity to Cache Size

CRUISE performance is dependent on the accuracy of the per-application cache utility. We evaluated CRUISE using profile based cache utility information on our baseline CMP system with smaller shared caches (2MB) and larger shared caches (8MB). With perfect cache utility knowledge, we found that CRUISE behaves similar to results illustrated in Figure 7 for all performance metrics. Similarly, when using RICE to dynamically classify applications, CRUISE behaves similar to results in Figure 9.

With regards to RICE-based dynamic utility classification, we found that classification of applications is a function of the available LLC size. For example, RICE correctly classifies sphinx as a LLC application on a 4MB and an LLC application on a 2MB cache. This is because sphinx has a roughly 4MB working-set size. Similarly, RICE correctly classifies k264ref as a LLC application on a 4MB cache but a LLC application on a 2MB cache. This is because k264ref has a roughly 2MB working-set size. In general, as expected, we observe transitions in application classification from cache fitting/friendly to cache thrashing/fitting when the LLC size is reduced. Similarly, we see transitions in application classification from cache thrashing/fitting to cache fitting/friendly as the LLC size is increased. This shows that RICE can dynamically classify the isolated application cache utility regardless of the available LLC size.

6. Related Work

Several researchers from the software and hardware community have independently studied and proposed ways to mitigate contention in shared caches. In this section we briefly describe recent work that is relevant to our proposal.

Perhaps the effort that closely resembles our work is the recent Distributed Intensity (DI) proposal [38]. Unlike our simulation based study, DI was evaluated on real hardware composed of Intel and AMD systems with shared LLCs. To the best of our knowledge, these LLCs use a pseudo-LRU replacement policy [10]. DI proposes to capture runtime cache miss rates of applications (using hardware performance counters), sort the miss rates, and then separate the memory intensive applications onto different LLCs, in effect co-scheduling memory intensive applications with non-memory bound applications. For an LRU managed cache, this co-scheduling strategy is at odds with all recent work on shared cache management [12, 25, 24, 8]. These studies have illustrated that in an LRU managed cache, memory intensive applications significantly degrade the performance of non-memory intensive applications. Perhaps the primary difference is because the authors correlate performance degradation to not only sharing the LLC but also from sharing prefetchers and the DRAM memory controller. More recent Intel and AMD processors, however, use a three-level hierarchy, where the prefetchers are private and located at the L2 cache [2]. Furthermore, the DRAM controller is on-die and more sophisticated than earlier designs [2]. Thus, the sharing effects of the prefetcher and DRAM controller is less pronounced in modern day processors with LRU replacement. As such, we show that DI does not perform well on LRU-managed caches. Furthermore, while CRUISE-D is similar to DI, it outperforms DI on DRRP-managed caches as well.

There has been considerable work in shared cache management via software cache partitioning [6, 17, 36]. The general idea in these designs is to allocate a portion of the cache to each of the applications and modify physical memory allocation such that every application’s cache lines map into its reserved cache space. These schemes have non-trivial and intricate interactions with the operating system kernel and require complicated changes to virtual memory management. Some software-centric scheduling algorithms attempt an ad-hoc approach to choosing which threads should be co-scheduled by attempting several thread allocation strategies and picking the best-performing ones [28]. This period during which the algorithm is sampling and learning can have sub-optimal system performance to dynamic phase changes. Furthermore, such a scheme is impractical for large number of threads since the number of permutations of co-schedules exponentially increases.

Recent work in understanding the software co-scheduling of threads has also shown that data sharing is an important parameter by which to classify threads [32]. Tang et al. focus on datacenter application workloads and use a heuristic based algorithm to predict which threads should be mapped to which cores. Since our runtime classification is based on miss rates it works in the same manner as a heuristic based algorithm.

More recent efforts attempt to design an intelligent scheduler that is aware of underlying caches and that can differentiate between applications with differing memory demands [7, 37, 38]. These proposals, however do not consider the impact that the underlying hardware replacement and allocation policies may have on the benefit of the scheduling decisions. Additionally, to differentiate between and classify the applications, these mechanisms do not take into account that hardware cache contention between the threads can lead to misguided classifications.

In the hardware management of shared caches, some of the earlier work focused on hardware cache partitioning [30, 25]. Hardware cache partitioning allocates cache resources to competing applications either statically or dynamically. These proposals use way partitioning in the cache and extra identifying bits per cache line to reserve portions of the cache for each application. Such designs work
well independently to manage caches at a hardware level but cannot reach optimal performance since they do not guide the software scheduler to make intelligent decisions in the first place.

More recent work in hardware management of caches has focused on smart insertion and replacement policies. These policies do not influence application co-schedules, but instead can mitigate the impact that thrashing and scans have on the performance of individual applications [12, 24].

The recent state-of-the-art work in smart hardware cache management focuses on more fine-grained classification of each application by differentiating between references that have near and distant reference intervals [10]. The RRIP and DRRIP proposals improve the performance of shared caches considerably and are good starting points to consider a hardware-software co-design where a smart cache replacement removes some of the inefficiencies of shared caches such that intelligent software scheduling is able to reach near optimal performance as we showed in this paper.

There has also been some independent work done in classifying the memory intensity of applications [5, 15, 35, 31]. These proposals draw analogies to differentiate between the memory requirements of several competing applications. However, the proposed mechanisms are more complex than our RICE proposal. As we show in this paper, RICE has negligible hardware overhead, and is practical design that can obtain online and dynamic classification information of concurrently executing applications.

7. Summary and Future Work

The use of virtualization, multi-core, and multi-socket systems has enabled multiple concurrently executing applications on the same system. In doing so, applications with varying memory demands contend for shared resources. Since the on-chip shared LLCs serves as the last-line of defense before the long-latency penalty to memory, it is crucial that shared LLCs be efficiently managed.

Shared LLCs commonly found in microprocessors today use the LRU replacement policy. Several studies have shown that LRU performs poorly for shared LLCs because LRU allocates cache resources based on demand instead of benefit. To address the LRU problem, shared LLCs have been managed independently by the software or the hardware. Software tries to intelligently co-schedule applications to avoid shared cache contention while hardware tries to reduce shared cache contention by improving the replacement policy. However, to-date there exists no comprehensive study that evaluates the interaction between improved cache replacement and intelligent scheduling decisions. With this in mind, this paper makes the following contributions:

- We conduct a detailed study on the interactions between intelligent scheduling and smart cache replacement. We find that smart cache replacement reduces the burden on software for intelligent scheduling but does not completely eliminate the need for finding optimal application co-schedules.

- We propose Cache Replacement and Utility-aware Scheduling (CRUISE), a hardware/software co-designed application scheduling policy that uses knowledge of the underlying LLC replacement policy and application cache utility information to determine how best to co-schedule applications.

- Finally, we propose a Runtime Isolated Cache Estimator (RICE), a hardware mechanism that dynamically determines isolated LLC performance while concurrently sharing the LLC with other applications. RICE requires no changes to the existing cache structure and requires storage overhead of only eight bytes per hardware-thread in the system.

For a large number of heterogeneous workload mixes, we evaluate CRUISE for a variety of systems with multiple shared LLCs. Our evaluations included LLCs shared by two and four applications. We show that CRUISE significantly reduces the performance variation between different static application co-schedules. In the majority of cases, CRUISE provides near-optimal performance for the throughput, weighted speedup, and fairness metrics.

In this paper, we evaluated CRUISE assuming as many cores as there are running applications. However, CRUISE is also applicable when the number of running applications is significantly larger than the number of cores. In such a system, the operating system or hypervisor can utilize CRUISE and RICE to dynamically select applications from the waiting queue while guaranteeing some degree of Quality of Service (QoS). Furthermore, we find that there is opportunity to reduce RICE overhead by dynamically controlling when to gather cache utility information. Exploring these extensions is part of our on-going work.

Acknowledgements

The authors would like to thank Bushra Ahsan, Malini Bhandaru, Jaideep Moses, Moinuddin Qureshi, Paul Racunas, Puneet Zaroo, and the anonymous reviewers for their feedback in improving the quality of this paper.

References


