A Compiler for Multiple Memory Models

Samuel. P. Midkiff *
IBM Thomas J. Watson Research Center

Jaejin Lee
Department of Computer Science and Engineering, Michigan State University
jlee@cs.msu.edu

David Padua
Department of Computer Science, University of Illinois
padua@uiuc.edu

Abstract

The design of consistency models for both hardware and software is a difficult task. For programming languages it is particularly difficult because the target audience for a programming language is much wider than the target audience for a hardware programming language, making usability a more important criteria. Exacerbating this problem is the reality that the programming languages community has little experience designing programming language consistency models, and therefore each new attempt is very much a voyage into uncharted territory. A concrete example of the difficulties of the task is the current Java Memory Model. Although designed to be easy to use by Java programmers, it is poorly understood and at least one common idiom (the "double check idiom") to exploit the model is unsafe. In this paper we describe the design of an optimizing Java compiler that will allow a consistency model for the code to be compiled to be specified as an input. The compiler will use Shasha and Snir's delay set analysis, and our CSSA program representation, to normalize the effects of different consistency models on optimizations and analysis. When completed, the compiler will serve as a testbed to prototype new memory models, and to measure the differences of different memory models on program performance.

1 Introduction and Problem Description

Consistency model design, both for hardware systems and programming languages, has long been a difficult and contentious topic. The difficulty of reaching a consensus on an ideal memory model is exacerbated by the fact that the quality of a model depends on both its ease of use (i.e. how hard it is to write correct programs using the model) and the performance the model can deliver to programs: two factors that seem to be inversely correlated.

In the area of software memory models, even comparisons of ease of use and performance are difficult. Ease of use comparisons are difficult because widely available implementations of different consistency models are targeted to a specific domain (e.g. technical computing[4]; system software and middleware[1], or general purpose computing[9]), and it is difficult to find different models implemented on the same machine for the same domain. It is even more difficult to find different models implemented on the same machine for general purpose computing. The same issues make performance comparisons difficult. And because compiler optimizations do not presently attempt inter-thread analysis, any performance numbers gathered may not be representative of the performance that the consistency model could obtain in that application.

Because popular software consistency models are tuned to existing compiler technologies, almost no experimental data exists for the relative costs of different consistency models in the presence of aggressive inter-thread analysis. This is particularly true for general purpose programs which drive the current computer market. Mark Hill[10, 11] offers some tantalizing numbers for hardware consistency models, using existing compiler technology with aggressive hardware design, that indicate that scientific codes running on sequentially consistent hardware may only suffer a 20% performance

*The corresponding author may be reached at smidkiff@us.ibm.com, or P.O. Box 218, Yorktown Heights, NY 10598.
This work is supported by NSF grant CCR-008125, and by the IBM Corporation.

penalty. We know of no such studies for general purpose programs or for consistency models that are part of the programming language semantics. And so, in the absence of data, the assumption is made that software consistency models must deviate significantly from sequential consistency – whose behavior is the target of all usable consistency models – to be efficient.

Into this information vacuum steps the widespread availability and adoption of multi-threaded programming via Java. Although Java is the first widely used programming language to define a multi-threaded consistency model as an integral part of the language[1], the importance of interactive event driven applications, networked applications that cannot freeze when waiting for network event, and multiprocessor hardware, it cannot be said that Java is an aberration. As is discussed later in more detail, the Java Memory Model (JMM) has tried provide both ease of use and performance using current compiler technology. The result has been a consistency (or memory) model with many problems, both in usability and ease of compilation. It is our belief that attempting to design usable, efficient consistency models within the current information vacuum is difficult, if not impossible. Moreover, given the subtlety of the trade-offs involved, producing a satisfactory consistency model without extensive prototyping and testing of various versions will not only be extremely difficult, but runs the risks of alienating the larger programming community before a satisfactory model is deployed. Finally, there is no reason to believe that "one size fits all", i.e. that there is one best consistency model.

This paper describes a flexible compilation system we are developing that will allow different consistency models to be easily prototyped, and their performance measured both with and without advanced inter-thread analysis algorithms. This approach will make the testing of new consistency models easier, allow the performance benefits of the consistency model relative to one another (including sequential consistency) to be measured; and will enlarge the design space of practical consistency models by letting optimizations make more usable models have acceptable performance. In short, we will provide a compilation system that allows compiler technology to bring performance and ease of use to memory models in a way analogous to how the software engineering benefits of object oriented programming and data abstraction are efficiently supported with advanced compiler technology. Because well-synchronized Java programs yield sequentially consistent execution outcomes, and the synchronization required is similar to that for other popular consistency models, we plan to use existing Java programs as the basis for building a benchmark suite.

The next section briefly outlines how the Java Memory Model has not lived up to its original goals, and serves to highlight the difficulties of memory model design, and some of the tradeoffs involved in memory model design. Section Section 3 describes some of the techniques we have developed, and will use, to analyze and optimize multi-threaded programs and conclude in Section Section 4.

2 An overview of problems with the Java Memory Model

The *Java Memory Model* differs from most previous consistency models in that it is defined as part of a language specification. In Java, this approach was meant to yield the result that all programs are well defined by the language (and associated virtual machine) specification alone, allowing portability and robustness across platforms. In contrast, a **C** program written for a *relaxed consistency* model, and ported to a *weak ordering* system, may require extensive reworking of the synchronization. Portability is an increasingly important requirement as economies of scale make the movement and deployment of large object frameworks across different hardware and software platforms desirable. Moreover, a single instance of an application may execute on a widely distributed system in which both the software and hardware are heterogeneous. We note that the software layer of TreadMarks provides many of the same benefits, albeit for a more restricted application domain than is typically targeted by Java.

In our opinion, the goals of the JMM were that (i) it be simple and understandable by a wide range of programmers; (ii) that it not adversely impact the performance of individual threads; and (iii) that it not require inter-thread analysis to generate correct and efficient programs. From interactions with others in the community of Java compiler and virtual machine developers, this opinion appears to be widespread. Despite the fact that the JMM was reviewed by a variety of experts in the programming languages community, it seems to have failed on each of these design goals.

(i) The JMM is *not* clear As has been extensively demonstrated in the JMM mailing list [12], and as anyone who has discussed the model with their colleagues can attest, the model is not simple. In fact, it has only been recently that two or more researchers working full time on Java might be able to confidently (and correctly) agree on the meaning of most of the model. This situation has improved somewhat because of extensive discussions about the model, but it is extremely unlikely that

[1] Other languages, including Ada, have done this, but they have never achieved the programmer base of Java. This is of great practical importance, because as the user base for a language expands, usability becomes more important.

this knowledge has, or will ever (because of the level of detail and complexity of the results) percolate down to practitioners – many of whom use Java because of its clariy and safeness

(ii) The JMM is *not* efficient Several ways the JMM adversely affects program performance have been known for some time, and were collected (and augmented) by Bill Pugh[23]. We will now describe a few of these, to show the extent of the problem.

The JMM requires that accesses to variables be coherent, i.e. memory operations on a single variable must be consistent. One consequence of this is that (naively) all reads of a variable must be ordered. If they are not ordered, then the following situation can arise. Let x be a shared variable. In one thread, the operations $x = 0; x = 1$ are performed. In another thread running in parallel with the first, the operations $y = x; z = x$ are performed. If the reads to x are rearranged, it is possible for $z = 0; y = 1$, an outcome not allowed by the JMM. Moreover, if the second thread consists of the operations $y = x; w = v; z = x$, and v and x are parameters that may be aliased, none of the reads can be reordered, since doing so may result in $z < y$, $z < w$, or $w < y$, none of which are allowed by the variable consistency required by the JMM. Moreover, in [23], a fairly complicated example is given of how memory operations between unaliased and non-volatile variables must be ordered, as well as a variety of other examples.

(iii) The JMM *requires* inter-thread analysis for efficiency Some of the problems with reads being ordered and dependences on the orders of unaliased memory accesses discussed in (ii) above require either a rewrite of the JMM, or inter-thread analysis to determine if reordering reads will not change the outcome of the program execution.

Our approach to this problem The computer science community should not be surprised that these (perhaps fatal) quirks have been uncovered. Mark Hill [10] has commented that even after working with memory models for many years, great care and thought must be given to answering almost any question because of the difficulty of understanding the application of these models. Although Prof. Hill was speaking of hardware models, the same caveats apply even more to consistency models for languages because of the wider user community, and the greater variation in skill levels of practitioners.

Our goal is to accelerate the development of the most usable and efficient programming language memory models possible. By building a compiler infrastructure that supports the analysis and optimization of explicitly parallel languages, we attack the performance side of the design equation, and increase the design space for memory models. By allowing programmable memory models, we give our system the flexibility to both rapidly prototype memory models – thereby shortening the design cycle for new memory models, and providing a mechanism for improving memory models for languages already in the field without invalidating existing programs written under an old memory model.

2.1 The benefits of programmable memory models

There are at least four major benefits of programmable memory models. The first is to allow rapid prototyping of new memory models within a compiler capable of applying aggressive optimizations to programs written with that memory model. This will allow apples-to-apples comparisons to be made between different memory models, since the compiler infrastructure for both is the same. It will also allow the performance of memory models to be tested in a framework that will support compilations with and without inter-thread analysis. Being able to measure the performance is important when evaluating memory models because the deviations from sequential consistency in all memory models known to the PI's is done to improve performance. Being able to measure performance with and without inter-thread analysis and optimization is important because it allows the value of inter-thread analysis to be measured, and it allows the actual cost of inter-thread analysis to be determined. The latter is particularly needed to determine the value of inter-thread analysis in dynamic compilation, systems.

The second benefit is to allow optimizers to be used by different programming languages. By making optimizations and analysis algorithms conform to the ordering required by a memory model, the optimizations and analysis are not limited to a single programming language or family of languages. Problems with retrofitting optimizations to both Java and OpenMP have been reported. In general these problems are overcome either by reducing the optimization level; by relying on optimizer conservatism when confronted with certain constructs, e.g. exceptions; and by not completely conforming to the language definition. Thus the current situation relies on stopgap measures.

The third benefit is to allow changes in language memory models without invalidating all programs previously written in the language. In a system like we propose, changing a language memory model would imply changing the default memory model. An option on the front-end that specifies the language version would change the default memory model, thereby causing the front-end to substitute the corresponding default model of the specified version for processing by the optimization and code-generation phases. This in turn will make it easier to get newer, improved memory models accepted by a language's user and vendor communities.

Fourth, because delay set analysis[24] (see Section 3.1), provides thread local information about the constraints on the ordering of memory accesses in the thread, and because delay set analysis assumes no single consistency model for the threads being analyzed, program optimization can take place across threads arising from different programming languages employing different consistency models. This is a powerful tool to support interoperability across programming languages.

Finally, programmable memory models will allow user specified memory models. We do not know how useful, or desirable, this will be, but our system will allow us examine this question. We note that although we are concentrating on consistency models that specify the observable behavior of memory accesses on different processors and threads, analogous techniques could be used to relax orders within a thread, or in a single threaded program. Thus, user programmable memory models might be used, for example, to specify independent regions to aid in compiler generated parallelism, prefetching, etc.

3 The analysis and optimization of languages with explicit parallelism and programmable memory models

We now show that the data-flow analysis and optimization of programs written in languages with programmable models is not only possible, but tractable. We first discuss *delay set analysis*[24], which forms the basis of our other analysis. We then describe algorithms for constructing efficient intermediate representations for explicitly parallel programs that represent the delay set analysis results and the sharing of data across threads; concurrent global value numbering (Section 3.3); common subexpression elimination (Section 3.4); and hoistable access detection (Section 3.5).

3.1 Delay set analysis

In [24], Shasha and Snir describe the minimal set of constraints necessary to honor in an execution such that the execution is legal. The analysis is conceptually simple – it seeks to find constraints (or program orderings) that if they were not enforced, would lead to program executions whose outcome is inconsistent with the constraints. Outcomes are inconsistent because they yields results, visible outside the program, that can only occur if a constraint is violated.

Consider the program and its graphical representation in Figure 3.1. Nodes in the graph represent memory accesses to shared data. Directed edges represent program orderings, or constraints. Undirected edges represent *conflicts* – accesses not necessarily ordered by constraints, but whose order can affect the outcome of the program. The *conflict* relation can be approximated by relating all accesses a_1, a_2 where a_1 and a_2 may reference the same storage, and either a is volatile, or at least one of a_1, a_2 is a write operation. We call the resulting graph a *program graph*.

We can *orient* the conflict edges by giving them a direction that represents the order of the two accesses during some execution of the program. A conservative solution to the delay set analysis problem is to find all *mixed* cycles, where a mixed cycle is one that contains both constraint and oriented conflict edges. These cycles represent inconsistent executions because no ordering of the edges in the graph exists which is consistent with the constraints in the program. The set of constraint arcs in the cycles are the (conservative approximation of) the set of constraints that it is necessary to honor to prevent the orientation of conflict edges that gives rise to the cycle. If the constraint arcs in this set are honored, then it is impossible at run-time for the an orientation of the conflict arcs to occur that gives rise to the inconsistent outcome[24]

The minimal solution given in [24] says that only the constraint arcs found in *minimal, mixed cycles* need to be enforced. A minimal mixed cycle C is one such that no cycle C' exists where the nodes of C' are a subset of the nodes of C. Intuitively, every non-minimal cycle must traverse the nodes of a minimal mixed cycle (or otherwise it would be minimal.) By enforcing the constraints in the minimal mixed cycle, it is impossible for conflict arcs to be oriented in such a way that either the minimal mixed cycle C', or the mixed cycle C, and the resulting inconsistent outcome, can exist at run-time.

3.1.1 Constructing program graphs

A program graph for delay set analysis can easily constructed. Nodes in the graph are labeled with symbols from a symbol set (described earlier), and an operation. For every pair of nodes n_1, n_2, we see if constraints specified by the consistency model applies to n_1, n_2 and the program order(s) between them. If there are one or more constraints that apply, the corresponding edge is placed on the graph.

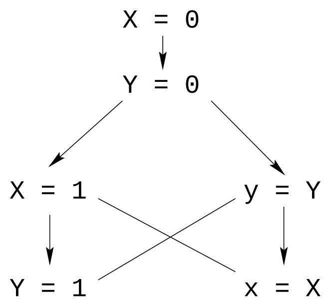

Figure 1: A delay set analysis example using sequential consistency. `y=1, x=0` is illegal.

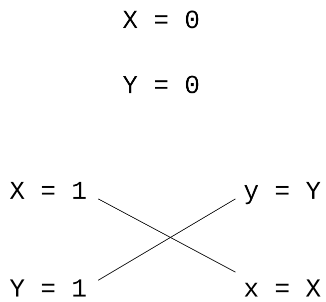

Figure 2: A delay set analysis example using JMM, with no aliasing and no exceptions. `y=1, x=0` allowed.

We note that our system will allow conservative construction of graphs – that is we allow more arcs than might strictly be placed according to the constraints. Cyclic constraints will be interpreted as in [24], that is, as atomic blocks. By not doing any optimization, it is always possible to generate a correct program, and therefore conservatism leads to correct programs.

Figure 3.1.1 shows the graph for the program of Figure 3.1 constructed with the constraints on variable coherence implied by the JMM. Fewer program arcs are shown than in the previous example because the Java Memory Model requires coherence only on the same variable. This allows the outcome $x = 0, y = 1$, which is inconsistent in sequential consistency, to be valid here because no cycle exists.

3.1.2 Performing delay set analysis

Although [24] state that a polynomial time algorithm exists for the delay set analysis problem, all exact published algorithms to date are exponential[16, 15, 21]. The algorithms of [16, 15] have been implemented, and on their set of benchmarks gave good compile time performance. For SPMD programs, Yelick and Krishnamurthy[14] have shown that arbitrary large numbers of threads can be approximately modeled as two threads, resulting in an exponent of two, and therefore a quadratic algorithm. The existing algorithms, seem, however, to be fast enough in practice on the programs they have been applied to [16, 15].

3.2 Intermediate Representations

In this section we briefly describe the concurrent control flow graph (CCFG) and the concurrent static single assignment (CSSA) form. The CSSA form is based on the CCFG. The CSSA form annotated with delay edges will be used as an intermediate representation in this project.

3.2.1 Concurrent Control Flow Graph

A *basic block* in a CCFG is a sequence of instructions where any branch is at the end, and there is at most one shared variable access. If there is more than one shared variable access in an assignment, we decompose the assignment into several assignments so that each contains at most one shared variable access. Within a basic block, we can apply classical optimization techniques safely without the need to consider interactions between threads because the accesses of shared variables are located at the beginning or at the end of the basic block, and the basic block contains at most one shared variable access. The shared variable access will be located at the beginning of a basic block if it is a read access, and at the end if it is a write access. Two basic blocks *conflict* if their variable accesses conflict. Conflict edges join conflicting basic blocks and are bidirectional. Each direction of a conflict edge has a label.

The label is δ^t if it is from a definition to a use, δ^a if it is from a use to a definition, and δ^o if it is from a definition to a definition. The labels appear at the head of the appropriate orientation of the conflict edge. Figure 4 shows the CCFG of the code in Figure 3. Threads are created at `cobegin` node and merged at `coend` node. A synchronization edge labeled σ goes from `post(S)` to a `wait(S)` on the event variable `S`. The region labeled $P3$ represents a `parallel do` construct which is represented by a `pdo` node, followed by the representation of the loop body, followed by an `endpdo` node. Similar to the control flow graph that is used by Krishnamurthy and Yelick[15, 16] for SPMD programs, we use two copies of the loop body of a `parallel do` construct when we construct a CCFG. One copy of the loop body contains conflict edges to itself for each loop carried dependence. Each statement in a copy of the loop body corresponds to a dual statement in the other copy. When we apply a compiler optimization to a `parallel do` construct, it must be applied to both dual statements symmetrically. If this is impossible, we do not apply the optimization.

3.2.2 Concurrent Static Single Assignment Form

The concurrent static single assignment form includes three functions; the ϕ-function that is the same as the one in SSA form[8], and two new functions, π and ψ. The SSA form contains ϕ-functions that represent the confluence of a variable from a group of its definitions, where each comes from a different incoming control flow edge. The ϕ-function in a CSSA form has the same meaning.

`Cobegin/coend` and `parallel do` constructs introduce new confluence points, e.g., `coend` and `endpdo` nodes. Instead of ϕ-functions, ψ-functions, which are similar to those defined by Srinivasan *et al.*[26, 27], are placed at these new confluence points and summarize the interleaving of defining assignments of shared variables in `cobegin/coend` and `parallel do` constructs.

The π-function summarizes the interleaving information of conflicting variables in the node where the π-function is placed. The π-function distinguishes values of shared variables reaching the point where the π-function is located due to the interleaving and synchronization of threads. Where do we need a π-function? Even though one thread modifies a shared variable, if another thread that runs concurrently with the thread does not use the variable, then the update of the shared variable does not affect the behavior of the other thread. Thus, we place a π-function of a shared variable at the point where there is a use of the variable only if there is an update of the shared variable by a different thread.

Like the SSA form for sequential programs, the CSSA form has the following properties:

- all uses of a variable are reached by exactly one (static) assignment to the variable
- for a variable, the definition dominates the uses if they are not arguments of ϕ-, ψ-, or π-functions

Translating a CCFG of a parallel program into its CSSA form is a four-step process:

1. Compute the execution ordering of concurrent basic blocks guaranteed by explicit synchronization.
2. Place ϕ-functions and ψ-functions.
3. Rename variables.
4. Place π-functions.

The algorithm for constructing a CSSA form is presented in detail in [20, 17]. In order to obtain information about the effect of interleaving statements in different threads, we need to compute the partial ordering of the basic blocks enforced by synchronization. Callahan and Subhlok[6] used data flow analysis to get an event ordering for explicitly parallel programs without loops. There also is a synchronization analysis technique developed by Callahan *et al.*[5] that can be applied to `parallel do` loops. It considers all the instances of statements in the iteration space and gives the ordering between them. A data flow analysis framework similar to theirs could be used to compute the ordering.

To insert ϕ- and ψ-assignments and rename variables, we can use the algorithm described by Cytron *et al.*[8]. At a `coend` and an `endpdo` nodes, ψ-assignments will be inserted instead of ϕ-assignments.

The ψ-function can be removed (before renaming) if the arguments are ordered. The argument with the lowest precedence kills the definitions of all other arguments and can be used instead of the ψ-function. A similar optimization is used in [26].

A reaching definition of a shared variable to its use may not be one of the arguments of the π-function for the use if it is killed by another reaching definition, or if its defining assignment executes after the use. We assign the π-function to a new temporary variable and insert this assignment immediately before the use. The use is replaced by this temporary variable.

The CSSA form of the CCFG in Figure 4 is shown in Figure 5.

```
s = 0
b = 0
f = 0
h = 0
i = 0
do while (s < 100)
  if (s > 70) then
    cobegin
      b = 3
      f = 1
      wait(g)
      t1 = s
      t2 = b
      s = ts + tb
    ||
      do while (f==0)
      end do
      b = 4
      post(g)
      h = s
    coend
  else
    parallel do i=1,4
      if (i==1) then
        b = 3
        wait(m)
        t3 = s
        t4 = b
        s = ts + tb
      else
        b = i
        post(m)
      end if
    end parallel do
    h = s
  end if
  print h
end do
```

Figure 3: A sample code.

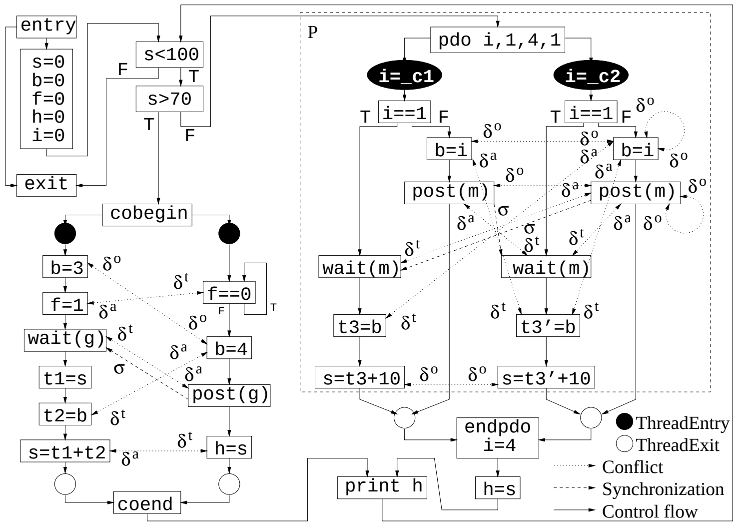

Figure 4: The concurrent control flow graph of the code in Figure 3.

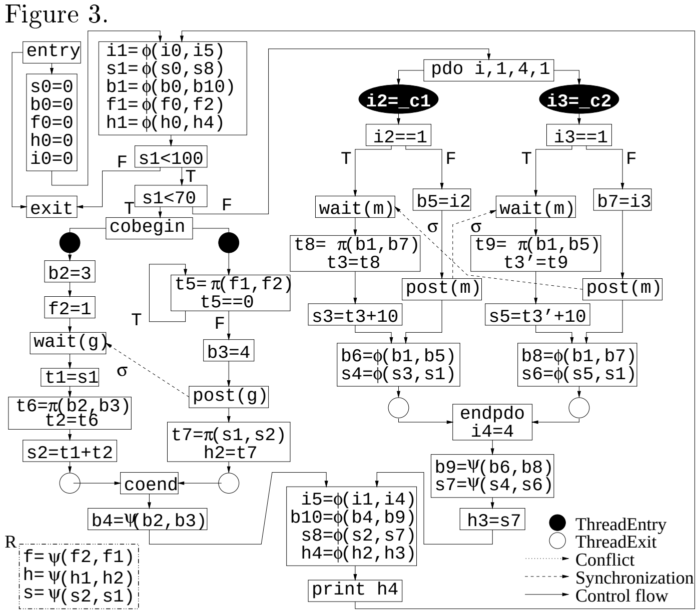

Figure 5: The CSSA form of the CCFG in Figure 4. For simplicity, conflict edges are omitted.

3.3 Concurrent Global Value Numbering

In this section, we describe an analysis technique called Concurrent Global Value Numbering (CGVN) that is required to implement the optimizing compiler. The value numbering technique has the property that it recognizes the equivalence of expressions even though the expressions are lexically different.

The equivalence of variables in sequential programs is defined by Alpern *et al.* in [3]. Two variables are said to be equivalent at a point p if those variables contain the same values whenever control reaches p during any possible execution of the program. Because the notion of equivalence contains dynamic instances of variables, Alpern *et al.* defined a congruence relation of nodes in a *value graph* to detect equivalence statically[3]. We use the same definition for equivalence of variables in parallel programs. However, the congruence of variables for parallel programs must consider the concurrency and interaction between threads.

As with sequential programs, we can construct a value graph[3] for a CSSA form and define the congruence relation on the value graph of a parallel program. Each node in the value graph corresponds to an assignment in the CSSA form. Each node has labels formed from the variable (variable label) defined in the assignment, and the operation (operation label) in the defining assignment. Each edge in the graph is from an operation node to the node that corresponds to an operand of the operation and labeled with the position of the corresponding operand in the operation. We can define congruence on the value graph of a parallel program as follows:

Definition 1 Congruence Relation on a Value Graph *Two nodes in the value graph are congruent if the following conditions hold:*

- *two nodes are congruent if their operation labels are the same and are not π or ψ, and their departing edges with the same labels lead to congruent nodes.*
- *a node with the operation label π or ψ is congruent to its arguments if all arguments are congruent to each other.*

Under these conditions, two nodes n_1 and n_2 with the operation label π (ψ) are congruent if one of the arguments of n_1 and one of the arguments of n_2 are congruent. The first condition is the same as that of the sequential case. To explain the reason behind the second condition, consider the following two π-assignments.

```
   ⋮
t0=π(a0,a1)
   ⋮
t1=π(b0,b1)
   ⋮
```

Assume congruence between `a0` and `b0` and between `a1` and `b1`. Also, assume the value of `t0` is `a1` in execution. Because the value of a π-function can be any of its arguments, the value of `t1` will not necessarily be that of `b1` in the execution. Thus, the first condition cannot be the condition that entails congruence between `t0` and `t1`. We assume a π-function or a ψ-function is congruent to its arguments when all the arguments are congruent. Intuitively, a π-function has the same value as its arguments if all the arguments have the same value. Because of the subscripts, two ϕ-functions in different CCFG nodes are never considered congruent. If the corresponding condition expression for a ϕ-function is known, the condition can be incorporated to determine congruence of ϕ-functions in different CCFG nodes[3].

The concurrent global value numbering (CGVN) algorithm is an optimistic partitioning algorithm to find congruent variables. The algorithm is an extension of a global value numbering algorithm for sequential programs that was developed by Cooper and Simpson[7, 25]. The global value numbering is essentially the same as Alpern *et al.*'s value graph partitioning[3]. To detect congruence between value graphs of two variables, we use an optimistic partitioning algorithm. All variables are assumed to be the same initially and this assumption is refined until a fixed point is reached. Similar to the sequential case[3], we can show that two congruent variables are equivalent at a point p if their defining assignments dominate p[17, 20].

3.4 Common Subexpression Elimination

Common subexpression elimination (CSE) is well known and one of the most common optimization techniques for sequential programs[22].

However, it cannot be applied directly to parallel programs. Consider the code in Figure 6(A). Because `a+k` in *S1* and `a+k` in *S2* are literally the same expressions, we can apply classical common

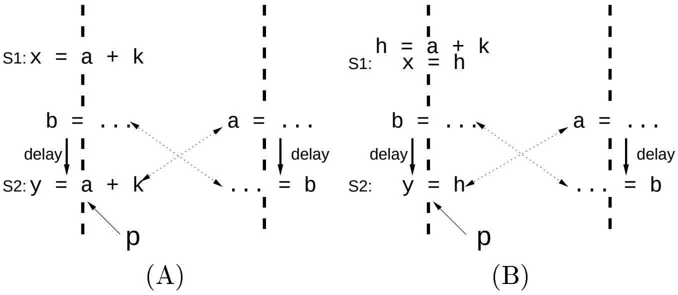

Figure 6: Sequential consistency violation by CSE.

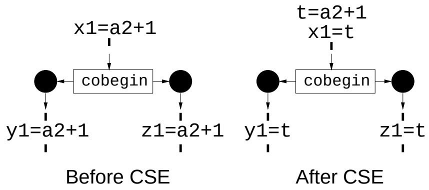

Figure 7: Unprofitable CSE.

subexpression elimination (CSE) to this code [Figure 6(B)]. Performing CSE in this case is equivalent to moving the statement *S2* to the point immediately after the statement *S1*, and executing it together with *S1*. This violates the delay constraints in Figure 6(A), and sequential consistency.

However, if we know that the variable `x` and `y` always have the same value at the point p in Figure 6(A), performing CSE will not violate sequential consistency even though it violates the delay constraints. Note that preserving delays are a sufficient condition to guarantee sequential consistency. To detect whether two variables are equivalent at a point, we will use concurrent global value numbering and dominance relation. As long as two variables are equivalent at a point p, and their defining assignments dominate the point p, we do not have to consider violations of sequential consistency when performing CSE.

As seen in Figure 7, performing CSE may introduce another shared variable access (`t` in Figure 7). This could be unprofitable[13] depending on the size of the common subexpression, because shared variable accesses take more time than local variable accesses. In some cases, doing CSE does not reduce or increase the execution time of a program. As seen in Figure 8, doing CSE in *Thread 1* will not reduce or increase the execution time of the program. However, doing CSE in the critical thread *Thread 2* will be beneficial.

Thus, in order to perform CSE in parallel programs, we must consider equivalence of two variables and profitability. For two congruent variables v and w, we apply CSE when the following conditions hold (S_v and S_w denote the defining assignments of v and w in the CSSA form):

- (equivalence condition) S_v dominates S_w.
- (profitability condition) application of CSE is profitable.

After CGVN is done on the CSSA form, CSE is applied to the CSSA form.

3.5 Hoistable Access Detection

Loop invariant code motion is another common optimization technique for sequential programs. Consider the code in Figure 9. The statement `a+1` in the `while` loop is a loop invariant in the classical sense [Figure 9(A)]. However, if we move the statement outside the loop [Figure 9(B)], it might make the loop infinite. This does not happen in the original program. For conventional compilers doing single threaded analysis, it is common to move a load instruction of a variable in a loop outside the loop if the load instruction is loop invariant. However, if we do not consider modification of the variable by another thread, moving load instructions is another source of correctness violations. We define a *hoistable access* for parallel programs that is a counterpart to a loop invariant in sequential programs.

Definition 2 *We call any computation in a loop that can be moved outside the loop without affecting the meaning of the given program a hoistable access.*

Similar to CSE, the notion of hoistable accesses also involves code motion. Thus, we must guard against sequential consistency violations. We mark a π-assignment in a loop *hoistable* if moving it outside the loop does not violate orderings required by delays, and if all its arguments are congruent to an argument that is a classical loop invariant. If all the variable accesses in an instruction in a loop are hoistable, then the instruction can be moved outside the loop.

3.6 Other Optimization Techniques

We described other classical optimization techniques that are problematic when directly applied to explicitly parallel programs, in [20, 17, 19, 18]. We also described their solution in these same papers. These techniques include constant propagation, copy propagation, redundant load/store elimination,

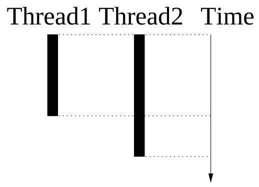

Figure 8: The execution time of a program depends on that of the critical thread Thread2.

```
a = 0
x = 0
cobegin
  do while (x == 2)
    x = a + 1
  end do
||
  a = 1
coend
```

(A)

```
a = 0
x = 0
cobegin
  t = a + 1
  do while (x == 2)
    x = t
  end do
||
  a = 1
coend
```

(B)

Figure 9: Incorrect loop invariant code motion.

and dead code elimination. Even though the solutions described in these papers, and in this section, assumed sequential consistency for the consistency model, the effects of the consistency model are encapsulated by the delay set analysis and the CSSA form, and the algorithms should work in the more general system we are proposing unchanged.

4 Conclusions

We have outlined the design of a compiler that will compile programs in which the memory model is a characteristic of the program and not the programming language. Such a compiler is desirable because it allows some key questions in the compilation of explicitly parallel languages to be answered, i.e.: (i) how useful is inter-thread analysis is to the optimization of different consistency models, and (iii) how expensive, in compiler execution time and space, are easy to use memory model than more performance oriented models, (ii) how expensive, in application execution time and space, are these simpler execution models. Moreover, the compiler will serves as a useful tool to prototype and test new consistency models.

We are constructing a compiler to implement these ideas. This compiler will build on the Jalapeno[2] compiler. The compiler will implement delay set analysis, the algorithms described in this proposal (and [20]), as well as new data-flow optimization and analysis algorithms developed. Algorithms to map from the language memory model to a storage memory model, have been developed and will be implemented.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, pages 66–76, December 1996.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeno virtual machine. *The IBM Systems Journal*, Feb. 2000. Available in PDF format at http://www.research.ibm.com/jalapeno/publication.html.

[3] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, pages 1–11, January 1988.

[4] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, pages 18–28, Feb 1996.

[5] David Callahan, Ken Kennedy, and Jaspal Subhlok. Analysis of event synchronization in a parallel programming tool. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 21–30, March 1990.

[6] David Callahan and Jaspal Subhlok. Static analysis of low-level synchronization. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 100–111, May 1988.

[7] Keith Cooper and Taylor Simpson. SCC-based value numbering. Technical Report CRPC-TR95636-S, Center for Research on Parallel Computation, Rice University, October 1995.

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, and Mark N. Wegman. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[9] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

[10] Mark D. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, pages 28–34, August 1998.

[11] Mark D. Hill. A retrospective on "Weak Ordering – A New Definition". In Gurindar S. Sohi, editor, *Selected Papers from the First 25 International Symposia on Computer Architecture*. ACM Press, 1998.

[12] Java memory model mailing list. Maintained by Bill Pugh, archives available at http://www.cs.umd.edu/ pugh/java/memoryModel/archive/.

[13] Jens Knoop and Bernhard Steffen. Code motion for explicitly parallel programs. In *Proceedings of The 1999 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–24, May 1999.

[14] Arvind Krishnamurthy and Katherine Yelick. Optimizing parallel SPMD programs. In *Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, August 1994.

[15] Arvind Krishnamurthy and Katherine Yelick. Optimizing parallel programs with explicit synchronization. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI)*, pages 196–204, June 1995.

[16] Arvind Krishnamurthy and Katherine Yelick. Analyses and optimizations for shared address space programs. *Journal of Parallel and Distributed Computing*, 38:139–144, 1996.

[17] Jaejin Lee. *Compilation Techniques for Explicitly Parallel Programs*. PhD thesis, University of Illinois at Urbana-Champaign, October 1999. Department of Computer Scienc Technical Report UIUCDCS-R-93-1814.

[18] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In Z. Li, P.-C. Yew, S. Chatterjee, C.-H. Huang, P. Sadayappan, and D. Sehr, editors, *Proceedings of The 10th International Workshop on Languages and Compilers for Parallel Computing*, number 1366 in Lecture Notes in Computer Science, pages 114–130. Springer, August 1997.

[19] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. A constant propagation algorithm for explicitly parallel programs. *International Journal of Parallel Programming*, 26(5):563–589, 1998.

[20] Jaejin Lee, David A. Padua, and Samuel P. Midkiff. Basic compiler algorithms for parallel programs. In *Proceedings of The 1999 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1999.

[21] S. P. Midkiff. Dependence analysis in parallel loops with $i \pm k$ subscripts. In C.H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *8'th International Workshop on Languages and Compilers for Parallel Computing, Columbus, Ohio.* Springer Verlag, Aug. 1995.

[22] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[23] William Pugh. Fixing the Java memory model. In *Proceedings of the ACM 1999 Java Grande Conference*, June 1999.

[24] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[25] Loren Taylor Simpson. *Value-Driven Redundancy Elimination*. PhD thesis, Department of Electrical and Computer Engineering, Rice University, April 1996.

[26] Harini Srinivasan and Dirk Grunwald. An efficient construction of parallel static single assignment form for structured parallel programs. Technical Report CU-CS-564-91, University of Colorado at Boulder, December 1991.

[27] Harini Srinivasan, James Hook, and Michael Wolfe. Static single assignment for explicitly parallel programs. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages (POPL)*, pages 260–272, January 1993.