Commutative Set: A Language Extension for Implicit Parallel Programming

Prakash Prabhu Soumyadeep Ghosh Yun Zhang Nick P. Johnson David I. August

Princeton University
Princeton, NJ
{pprabhu, soumyade, yunzhang, npjohnso, august}@princeton.edu

Abstract

Sequential programming models express a total program order, of which a partial order must be respected. This inhibits parallelizing tools from extracting scalable performance. Programmer written semantic commutativity assertions provide a natural way of relaxing this partial order, thereby exposing parallelism implicitly in a program. Existing implicit parallel programming models based on semantic commutativity either require additional programming extensions, or have limited expressiveness. This paper presents a generalized semantic commutativity based programming extension, called Commutative Set (COMMSET), and associated compiler technology that enables multiple forms of parallelism. COMMSET expressions are syntactically succinct and enable the programmer to specify commutativity relations between groups of arbitrary structured code blocks. Using only this construct, serializing constraints that inhibit parallelization can be relaxed, independent of any particular parallelization strategy or concurrency control mechanism. COMMSET enables well performing parallelizations in cases where they were inapplicable or non-performing before. By extending eight sequential programs with only 8 annotations per program on average, COMMSET and the associated compiler technology produced a geomean speedup of 5.7x on eight cores compared to 1.5x for the best non-COMMSET parallelization.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming; D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization

General Terms Languages, Performance, Design, Experimentation

Keywords Implicit parallelism, semantic commutativity, programming model, automatic parallelization, static analysis

1. Introduction

The dominant parallel programming models for multicores today are explicit [8, 10, 33]. These models require programmers to expend enormous effort reasoning about complex thread interleavings, low-level concurrency control mechanisms, and parallel programming pitfalls such as races, deadlocks, and livelocks. Despite this effort, manual concurrency control and a fixed choice of parallelization strategy often result in parallel programs with poor performance portability. Consequently, parallel programs often have to be extensively modified when the underlying parallel substrates evolve, thus breaking abstraction boundaries between software and hardware.

Recent advances in automatic thread extraction [26, 30, 34] provide a promising alternative. They avoid pitfalls associated with explicit parallel programming models, but retain the ease of reasoning of a sequential programming model. However, sequential languages express a total order, of which a partial order of program execution must be respected by parallelizing tools. This prohibits some execution orders that are often permitted by high-level algorithm specifications. As a result, automatic parallelizing tools, overly constrained by the need to respect the sequential semantics of programs written in languages like C/C++, are unable to extract scalable performance.

Implicit parallel programming models (IPP) [4, 13, 15, 31] offer the best of both approaches. In such models, programmers implicitly expose parallelism inherent in their program without the explicit use of low-level parallelization constructs. An interesting subclass of models within this space includes those that are based on top of sequential programming models [15]. In such models, programmer insights about high-level semantic properties of the program are expressed via the use of extensions to the sequential model. These language extensions are then exploited by transformation tools to automatically synthesize a correct parallel program. This approach not only frees the programmer from the burden of having to worry about the low-level details related to parallelization, but also promotes retargetability of such programs when presented with newer parallel substrates.

Recent work has shown the importance of programmer specified semantic commutativity assertions in exposing parallelism implicitly in code [5, 7, 18, 27, 30]. The programmer relaxes the order of execution of certain functions that read and modify mutable state, by specifying that they legally *commute* with each other, despite violating existing partial orders. Parallelization tools exploit this relaxation to extract performance by permitting behaviors prohibited under a sequential programming model. However, existing solutions based on semantic commutativity either have limited expressiveness or require programmers to use additional parallelism constructs. This paper proposes an implicit parallel programming model based on semantic commutativity, called Commutative Set (COMMSET), that generalizes existing semantic commutativity

PLDI'11, June 4–8, 2011, San Jose, California, USA.
Copyright © 2011 ACM 978-1-4503-0663-8/11/06. . . $10.00

IPP System	Concept					Specific Parallel Implementation					
	Expressiveness of Commutativity Specification				Requires Additional Extensions	Parallelism Forms Supported			Concurrency Control Mechanism	Parallelization Driver	Optimistic or Speculative Parallelism[1]
	Predication		Commuting Blocks	Group Commutativity		Task[1]	Pipelined	Data			
	Interface	Client									
Jade [27]	×	×	×	×	Yes	✓	✓	×	Automatic	Runtime	×
Galois [18]	✓	×	×	×	Yes	×	×	✓	Manual	Runtime	✓
DPJ [5]	×	×	×	×	Yes	✓	×	✓	Manual	Programmer	×
Paralax [30]	×	×	×	×	No	×	✓	×	Automatic	Compiler	×
VELOCITY [7]	×	×	×	×	No	×	✓	×	Automatic	Compiler	✓
CommSet	✓	✓	✓	✓	No	×	✓	✓	Automatic	Compiler	×

Table 1: Comparison between CommSet and other parallel models based on semantic commutativity

constructs and enables multiple forms of parallelism from the same specification.

Table 1 compares CommSet with existing parallel programming models based on semantic commutativity. The main advantages of CommSet over existing approaches are: (a) CommSet's commutativity construct is more general than others. Prior approaches allow commutativity assertions only on interface declarations. However, commutativity can be a property of client code as well as code behind a library interface. CommSet allows the commutativity assertions between arbitrary structured code blocks in client code as well as on interfaces, much like the `synchronized` keyword in Java. It also allows commutativity to be predicated on variables in a client's program state, rather than just function arguments as in earlier approaches. (b) CommSet specifications between a group of functions are syntactically succinct, having linear specification complexity rather than quadratic as required by existing approaches. (c) CommSet presents an implicit parallel programming solution that enables both pipeline and data parallelism without requiring any additional parallelism constructs. Existing approaches use parallelism constructs that tightly couple parallelization strategy with concrete program semantics, in contravention of the principle of "separation of concerns." In contrast, using only CommSet primitives in our model, parallelism can be implicitly specified at a semantic level and is independent of a specific form or concurrency control mechanism.

The contributions of this work are:

1. The design, syntax, and semantics of CommSet, a novel programming extension that generalizes, in a syntactically succinct form, various existing notions of semantic commutativity.
2. An end-to-end implementation of CommSet within a parallelizing compiler that includes the front-end, static analysis to enhance the program dependence graph with commutativity properties, passes to enable data and pipeline parallelizations, and automatic concurrency control.
3. A demonstration of CommSet's applicability in expressing implicit parallelism by extending eight real world sequential programs with commutativity assertions, and an evaluation of the performance on real hardware.

The features of CommSet are first motivated by a running example. A description of its syntax and semantics follows. An implementation of CommSet within a parallelizing compiler is explained step by step, followed by an evaluation and a discussion of related work.

2. Motivating Example

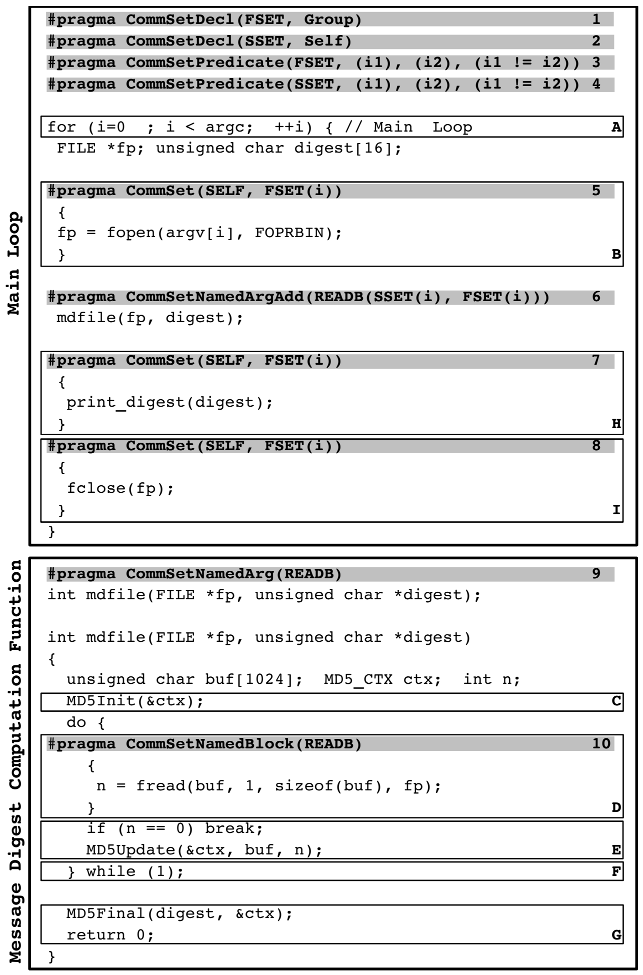
Figure 1 shows a code snippet from a sequential implementation of md5sum (plus highlighted `pragma` directives introduced for CommSet that are discussed later). The main loop iterates through a set of input files, computing and printing a message digest for each file. Each iteration opens a file using a call to `fopen`, then calls the `mdfile` function which, in turn, reads the file's contents via calls to `fread` and then computes the digest. The main loop prints the digest to the console and closes the file by calling `fclose` on the file pointer. Although it is clear that digests of individual files can be safely computed out of order, a parallelizing tool cannot infer this automatically without knowing the client specific semantics of I/O calls due to its externally visible side effects. However, the loop can be parallelized if the commuting behaviors of `fopen`, `fread`, `fclose`, and `print_digest` on *distinct* files are conveyed to the parallelizing tool.

One way to specify commuting behavior is at the interface declarations of file operations. Galois [18] extracts optimistic parallelism by exploiting semantic commutativity assertions specified between pairs of library methods at their interface declarations. These assertions can optionally be predicated on their arguments. To indicate the commuting behaviors of the calls on distinct files, one would ideally like to predicate commutativity on the filename. Since only `fopen` takes in the filename as an argument, this is not possible. Another approach is to predicate commutativity on the file pointer `fp` that is returned by `fopen`. Apart from the fact that expensive runtime checks are required to validate the assertions *before* executing the I/O calls (which are now on the critical path), this approach may prevent certain valid commuting orders due to recycling of file pointers. Operations on two distinct files at different points in time that happen to use the same file pointer value are now not allowed to commute. This solution is also not valid for all clients. Consider the following sequence of calls by a client that writes to a file (`fp1`) and subsequently reads from it (`fp2`) in the next iteration: `fwrite(fp1)`, `fclose(fp1)`, `fopen(fp2)`, `fread(fp2)`. Here, even though `fp1` and `fp2` may have different runtime values, they still may be pointing to the same file. Commuting `fopen(fp2)`, `fread(fp2)` with `fclose(fp1)` may cause a read from the file before the write file stream has been completed. Approaches that annotate data (file pointer `fp` in this case) to implicitly assert commutativity between all pairs of operations on that file pointer [27], run into the same problem.

Allowing predication on the client's program state can solve the above problem for md5sum. Since the input files naturally map to different values of the induction variable, predicating commutativity on the induction variable (in the client) solves the problems associated with interface based predication. First, no legal commuting behavior is prohibited since induction variables are definitely different on each iteration. Second, runtime checks for commutativity assertions are avoided since the compiler can use static analysis to symbolically interpret predicates that are functions of the induction variable to prove commutativity on separate iterations.

In order to continue using commutativity specifications on function declarations while still predicating on variables in client state, programmers either have to change existing interfaces or create new wrapper functions to take in those variables as arguments. Changing the interface breaks modularity since other clients which do not want commutative semantics are now forced to pass in ad-

[1] The commutativity specifications languages of Galois, Paralax, VELOCITY and CommSet are conceptually amenable to task and speculative parallelism



```
#pragma CommSetDecl(FSET, Group)                                  1
#pragma CommSetDecl(SSET, Self)                                   2
#pragma CommSetPredicate(FSET, (i1), (i2), (i1 != i2))            3
#pragma CommSetPredicate(SSET, (i1), (i2), (i1 != i2))            4

for (i=0  ; i < argc;  ++i) { // Main  Loop                       A
 FILE *fp; unsigned char digest[16];

#pragma CommSet(SELF, FSET(i))                                    5
 {
 fp = fopen(argv[i], FOPRBIN);
 }                                                                B

#pragma CommSetNamedArgAdd(READB(SSET(i), FSET(i)))               6
 mdfile(fp, digest);

#pragma CommSet(SELF, FSET(i))                                    7
 {
  print_digest(digest);
 }                                                                H
#pragma CommSet(SELF, FSET(i))                                    8
 {
  fclose(fp);
 }                                                                I
}

#pragma CommSetNamedArg(READB)                                    9
int mdfile(FILE *fp, unsigned char *digest);

int mdfile(FILE *fp, unsigned char *digest)
{
  unsigned char buf[1024];  MD5_CTX ctx;  int n;
  MD5Init(&ctx);                                                  C
  do {
#pragma CommSetNamedBlock(READB)                                  10
    {
     n = fread(buf, 1, sizeof(buf), fp);
    }                                                             D
    if (n == 0) break;
    MD5Update(&ctx, buf, n);                                      E
  } while (1);                                                    F

  MD5Final(digest, &ctx);
  return 0;                                                       G
}
```

Figure 1: Sequential version of md5sum extended with COMMSET

ditional dummy arguments to prevent commuting behaviors. Creating wrapper functions involves additional programmer effort, especially while replicating functions along entire call paths. In the running example, the `mdfile` interface has to be changed to take in the induction variable as an argument, to allow for predicated commutativity of `fread` calls with other file operations (`fopen` and `fclose`) in the main loop.

The additional programmer effort in creating wrappers can be avoided by allowing structured code blocks enclosing the call sites to commute with each other. This is easily achieved in md5sum by enclosing the call sites of `fopen`, `fread`, `print_digest`, and `fclose` within anonymous commutative code blocks. Commutativity between multiple anonymous code blocks can be specified easily by adding the code blocks to a named set, at the beginning of their lexical scope. Grouping commutative code blocks or functions into a set, as presented here, has linear specification complexity. In contrast, existing approaches [5, 18] require specifying commutativity between pairs of functions individually leading to quadratic specification complexity. The modularity problem (mentioned above) can be solved by allowing optionally commuting code blocks and exporting the option at the interface, without changing interface arguments. Clients can then enable the commutativity of the code blocks if it is in line with their intended semantics, otherwise default sequential behavior is preserved. In the case of md5sum, the `fread` call can be made a part of an optionally commuting block. The option is exported at the interface declara-

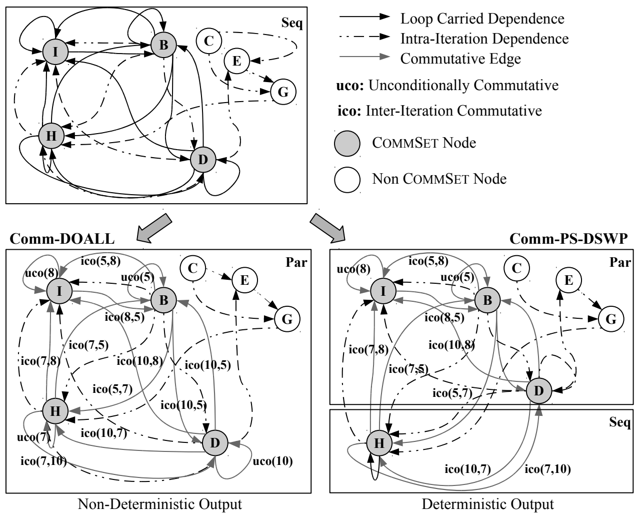


Figure 2: PDG for md5sum with COMMSET extensions

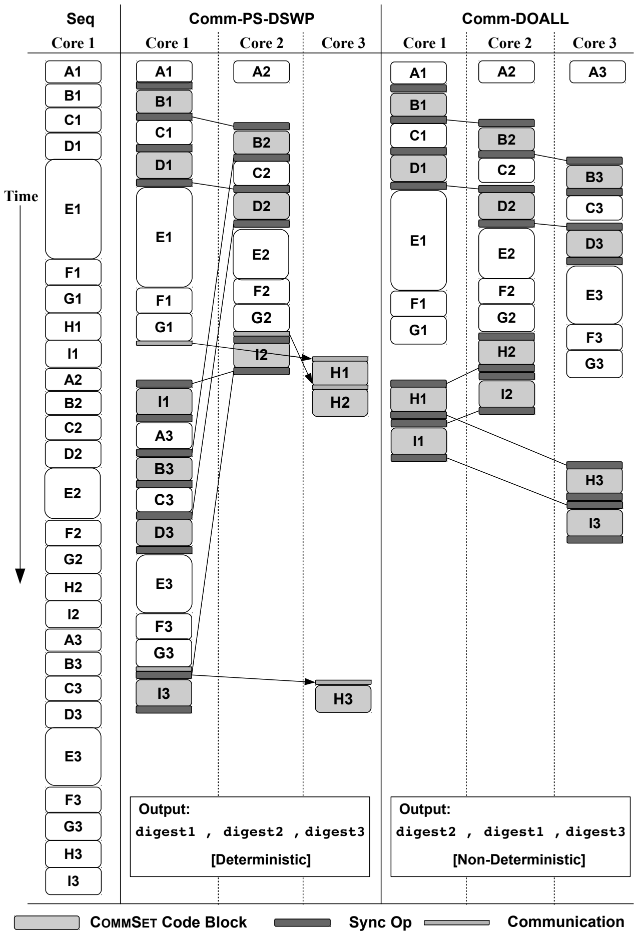


Figure 3: Timeline for md5sum Parallelizations

tion of `mdfile` and is enabled by the main loop, while other clients that require sequential semantics can ignore the option.

A commuting code block gives the programmer the flexibility to choose the extent to which partial orders can be relaxed in a program. This, in turn, determines the amount of freedom a parallelizing tool has, to extract parallelism. For instance, enclosing `fread` calls inside `mdfile` within a commuting block gives more freedom to a parallelizing tool to extract performance, as opposed to the outer call to `mdfile`. Depending on the intended semantics, the programmer can choose the right granularity for the commuting blocks which a parallelizing system should automatically guarantee to be atomic. Commuting blocks allow for a declarative specification of concurrency which a parallelizing tool can exploit to automatically select the concurrency mechanism that performs best for a given application. Automatic concurrency control has the advantage of not requiring invasive changes to application code when newer mechanisms become available on a particular hardware substrate.

Parallelizing tools should be able to leverage the partial orders specified via commutativity assertions without requiring the programmer to specify parallelization strategies. Existing runtime approaches require the use of additional programming extensions that couple parallelization strategies to program semantics. Galois [18] requires the use of set iterators that constrain parallelization to data parallelism. Jade [27] requires the use of task parallel constructs. DPJ [5] uses explicitly parallel constructs for task and data parallelism. COMMSET does not require such additional extensions. For instance, in md5sum, a programmer requiring deterministic output for the digests, should be able to express the intended semantics without being concerned about parallelization. The implementation should be able to automatically change to the best parallelization strategy given the new semantics. Returning to md5sum, specifying that `print_digest` commutes with the other I/O operations, but not with itself constrains output to be deterministic. Given the new semantics, the compiler automatically switches from a better performing data parallel execution of the loop to a slightly less performing (in this case) pipelined execution. In the data parallel execution, each iteration of the loop executes in parallel with other iterations. In a pipeline execution, an iteration is split into stages, with the message digests computed in parallel in earlier stages of the pipeline being communicated to a sequential stage that prints the digest in order to the console.

Parallelization within the compiler is based on the Program Dependence Graph (PDG) structure [12]. Figure 2 shows the simplified PDG for sequential md5sum. Each labeled code block is represented by a node in the PDG and a directed edge from a node n_1 to n_2 indicates that n_2 is dependent on n_1. Parallelizing transforms (e.g. DOALL [16] and PS-DSWP [26]) partition the PDG and schedule nodes onto different threads, with dependences spanning threads automatically respected by insertion of communication and/or synchronization operations. With the original PDG, DOALL and PS-DSWP cannot be directly applied due to a cycle with loop carried dependences: $B \rightarrow D \rightarrow H \rightarrow I \rightarrow B$ and self loops around each node in the cycle. The COMMSET extensions help the compiler to relax these parallelism-inhibiting dependences, thereby enabling wider application of existing parallelizing transforms.

Figure 3 shows three schedules with different performance characteristics for md5sum execution. The first corresponds to sequential execution and the other two parallel schedules are enabled by COMMSET. Each of these schedules correspond to three different semantics specified by the programmer. The sequential execution corresponds to the in-order execution of all I/O operations, as implied by the unannotated program. The PS-DSWP schedule corresponds to parallel computation of message digests overlapped with the sequential in-order execution of `print_digest` calls. Finally, the DOALL schedule corresponds to out-of-order execution of digest computation as well `print_digest`s. Every COMMSET block in both the DOALL and PS-DSWP schedules is synchronized by the use of locks (provided by libc), while PS-DSWP has additional communication operations. The DOALL schedule achieves a speedup of 7.6x on eight threads over sequential execution while PS-DSWP schedule gives a speedup of 5.8x. The PS-DSWP schedule is the result of one less COMMSET annotation than the DOALL schedule. The timeline in Figure 3 illustrates the impact of the semantic choices a programmer makes on the freedom provided to a parallelizing tool to enable well performing parallelizations. In essence, the COMMSET model allows a programmer to concentrate on high-level program semantics while leaving the task of determining the best parallelization strategy and synchronization mechanism to the compiler. In doing so, it opens up an parallel performance optimization space which can be systematically explored by automatic tools.

3. Syntax and Semantics

This section describes the semantics of various COMMSET features and the syntax of the COMMSET primitives.

3.1 CommSet Semantics

Self and Group Commutative Sets. The simplest form of semantic commutativity is a function commuting with itself. A *Self* COMMSET is defined as a singleton set with a code block that is self-commutative. An instantiation of this COMMSET allows for reordering dynamic invocation sequences of the code block. A straightforward extension of self-commutativity that allows commutativity between pairs of functions has quadratic specification complexity. Grouping a set of commuting functions under a name can reduce the specification burden. However, it needs to account for the case when a function commutes with other functions, but not with itself. For instance, the `insert()` method in a STL `vector` implementation does not commute with itself, but commutes with `search()` on different arguments. A *Group* COMMSET is a set of code blocks where pairs of blocks commute with each other, but each block does not commute with itself. These two concepts together achieve the goal of completeness and conciseness of commutativity specification.

Domain of Concurrency. A COMMSET aggregates a set of code blocks that read and modify shared program state. The members are executed concurrently in a larger parallelization scope, with atomicity of each member of the COMMSET guaranteed by automatic insertion of appropriate synchronization primitives. In this sense, COMMSET plays the role of a "concurrency domain," with updates to the shared mutable state being done in arbitrary order. However, the execution orders of members of a COMMSET with respect to the rest of code (sequential or other COMMSETs) are determined by flow dependences present in sequential code.

Non-transitivity and Multiple Memberships. Semantic commutativity is intransitive. Given three functions `f`, `g`, and `h` where two pairs (`f`, `g`), (`f`, `h`) semantically commute, the commuting behavior of (`g`, `h`) cannot be automatically inferred without knowing the semantics of state shared exclusively between `g` and `h`. Allowing code blocks to be members of multiple COMMSETs enables expression of either behavior. Programmers may create a single COMMSET with three members or create two COMMSETs with two members each, depending on the intended semantics. Two code blocks commute if they are both members of *at least* one COMMSET .

Commutative Blocks and Context Sensitivity. In many programs that access generic libraries, commutativity depends on the client code's context. The COMMSET construct is flexible enough to allow for commutativity assertions at either interface level or in client code. It also allows arbitrary structured code blocks to commute with other COMMSET members, which can either be functions or structured code blocks themselves. Functions belonging to a module can export optional commuting behaviors of code blocks in their body by using named block arguments at their interface, without requiring any code refactoring. The client code can choose to enable the commuting behavior of the named code blocks at its call site, based on its context. For instance, the `mdfile` function in Figure 1 that has a code block containing calls to `fread` may expose the commuting behavior of this block at its interface via the use of a named block argument *READB*. A client which does not care about the order of `fread` calls can add *READB* to a COMMSET optionally at its call site, while clients requiring sequential order can ignore the named block argument. Optional COMMSET specifications do not require any changes to the existing interface arguments or its implementation.

Predicated Commutative Set. The COMMSET primitive can be predicated on either interface arguments or on variables in a client's program state. A predicate is a C expression associated with the COMMSET and evaluates to a Boolean value when given arguments corresponding to any two members of the COMMSET . The predicated expression is expected to be pure, i.e. it should return the same value when invoked with the same arguments. A pure predicate expression always gives a deterministic answer for deciding commutativity relations between two functions. The two members commute if the predicate evaluates to true when its arguments are bound to values of actual arguments supplied at the point where the members are invoked in sequential code.

Orthogonality to Parallelism Form. Semantic commutativity relations expressed using COMMSET are independent of the specific form of parallelism (data/pipeline/task) that are exploited by the associated compiler technology. Once COMMSET annotations are added to sections of a sequential program, the same program is amenable to different styles of parallelization. In other words, a single COMMSET application can express commutativity relations between (static) lexical blocks of code and dynamic instances of a single code block. The former implicitly enables pipeline parallelism while the latter enables data and pipeline parallelism.

Well-defined CommSet members. The structure of COMMSET members has to obey certain conditions to ensure well-defined semantics in a parallel setting, especially when used in C/C++ programs that allow for various unstructured and non-local control flows. The conditions for ensuring well-defined commutative semantics between members of a COMMSET are: (a) The control flow constructs within each code block member should be local or structured, i.e. the block does not include operations like `longjmp`, `setjmp`, etc. Statements like `break` and `continue` should have their respective parent structures within the commutative block. (b) There should not be a transitive call from one code block to another in the same COMMSET . Removing such call paths between COMMSET members not only avoids the ambiguity in commutativity relation defined between a caller and a callee, but also simplifies reasoning about deadlock freedom in the parallelized code. Both these conditions are checked by the compiler.

Well-formedness of CommSets. The concept of well-definedness can be extended from code blocks in a single COMMSET to multiple COMMSETs by defining a COMMSET graph as follows: A COMMSET *graph* is defined as a graph where there is a unique node for each COMMSET in the program, and there exists an edge from a node S_1 to another node S_2, if there is a transitive call in the program from a member in COMMSET S_1 to a member in COMMSET S_2. A set of COMMSETs is defined to be *well-formed* if each COMMSET has well-defined members and there is no cycle in the COMMSET graph. Our parallelization system guarantees deadlock freedom in the parallelized program if the only form of parallelism in the input program is implicit and is expressed through a well-formed set of COMMSETs. This guarantee holds when either pipeline or data parallelism is extracted and for both pessimistic and optimistic synchronization mechanisms (see Section 4.6).

CommSet Global Declarations	`#pragma` **`CommSetDecl`**`(CSet,` **`SELF`**`|`**`Group`**`)`
	
	`#pragma` **`CommSetPredicate`**`(CSet,`
	(x_1, ..., x_n), (y_1, ..., y_n),
	Pred(x_1, ..., x_n, y_1, ..., y_n))
	
	`#pragma` **`CommSetNoSync`**`(CSet)`

CommSet Instance Declarations	`#pragma` **`CommSet`**`(CSets)`
	`|`**`CommSetNamedArg`**`(Blocks)`
	$\texttt{type}_{\texttt{return}}$ function-name(t_1 x_1, ..., t_n x_n);
	
	`#pragma` **`CommSet`**`(CSets)`
	`|`**`CommSetNamedBlock`**($B_{\texttt{name}}$)
	`{`
	`Structured code region`
	`}`
	
	`#pragma` **`CommSetNamedArgAdd`**`(BPairs)`
	retval = function-name(x_1, ...,x_n);

CommSet List	`CSets     ::= PredCSet | PredCSet, CSets`
	`PredCSet ::= CSet | CSet(CVars)`
	CVars ::= $C_{\texttt{var}}$ | $C_{\texttt{var}}$, Cvars
	CSet ::= **SELF** | $\texttt{Set}_{\texttt{id}}$
	BPairs ::= $B_{\texttt{id}}$(CSets) |$B_{\texttt{id}}$(CSets), BPairs
	Blocks ::= $B_{\texttt{id}}$ | $B_{\texttt{id}}$, Blocks

Figure 4: COMMSET Syntax

3.2 CommSet Syntax

The COMMSET extensions are expressed using `pragma` directives in the sequential program. `pragma` directives were chosen because a program with well-defined sequential semantics is obtained when they are elided. Programs with COMMSET annotations can also be compiled without any change by a standard C/C++ compiler that does not understand COMMSET semantics.

Global Declarations. The name of a COMMSET is indicative of its type. By default, the SELF keyword refers to a Self COMMSET, while COMMSETs with other names are Group COMMSETs. To allow for predication of Self COMMSETs, explicit type declaration can be used. The COMMSETDECL primitive allows for declaration of COMMSETs with arbitrary names at global scope. The COMMSETPREDICATE primitive is used to associate a predicate with a COMMSET and is declared at global scope. The primitive takes as arguments: (a) the name of the COMMSET that is predicated, (b) a pair of parameter lists, and (c) a C expression that represents the predicate. Each parameter list represents the subset of program state that decides the commuting behavior of a pair of COMMSET members, when they are executed in two different parallel execution contexts. The parameters in the lists are bound to either a commutative function's arguments, or to variables in the client's program state that are live at the beginning of a structured commutative code block. The C expression computes a Boolean value using the variables in the parameter list and returns true if a pair of COMMSET members commute when invoked with the appropriate arguments. By default, COMMSET members are automatically synchronized when their source code is available to the parallelizing compiler. A programmer can optionally specify that a COMMSET does not need compiler inserted synchronization using COMMSETNOSYNC. The primitive is applied to COMMSETs whose members belong to a thread-safe library which has been separately compiled and whose source is unavailable.

Instance Declarations and CommSet List. A code block can be declared a member of a list of COMMSETs by using the COMMSET directive. Such instance declarations can be applied either at a function interface, or at any point in a loop or a function for adding an arbitrary structured code block (a compound statement in C/C++) to a COMMSET. Both compound statements and functions are treated in the same way as far as reasoning about commutativity is concerned. In the case of predicated COMMSETs in the COMMSET list, the actual arguments for the COMMSETPREDICATE are supplied at the instance declaration. For function members of a COMMSET, the actual arguments are a list of parameter declarations, while for compound statements, the actual arguments are a set of variables with primitive type that have a well-defined value at the beginning of the compound statement. Optionally commuting compound statements can be given a name by enclosing the statements within COMMSETNAMEDBLOCK directive. A function containing such a named block can expose the commuting option to client code using COMMSETNAMEDARG at its interface declaration. The client code that invokes the function can enable the commuting behavior of the named block by adding it to a COMMSET using the COMMSETNAMEDARGADD directive at its call site.

3.3 Example

Figure 1 shows the implicitly parallel program obtained by extending md5sum with COMMSET primitives. The code blocks *B*, *H*, *I* enclosing the file operations are added to a *Group* COMMSET *FSET* using annotations *5*, *7*, and *8*. Each code block is also added to its own *Self* COMMSET. *FSET* is predicated on the loop induction variable's value, using a COMMSETPREDICATE expression (*3*) to indicate that each of the file operations commute with each other on separate iterations. The block containing `fread` call is named *READB* (*10*) and exported by `mdfile` using the COMMSETNAMEDARG directive at its interface (*9*). The client code adds the named block to its own *Self* set (declared as *SSET* in *2*) using the COMMSETNAMEDARGADD directive at *6*. *SSET* is predicated on the outer loop induction variable to prevent commuting across inner loop invocations (*4*). A deterministic output can be obtained by omitting *SELF* from annotation *7*.

4. Commutative Set Implementation

We built an end-to-end implementation of COMMSET within a parallelizing compiler. The compiler is an extension of the clang/LLVM framework [19]. Figure 5 shows the parallelization workflow. The parallelization focuses on hot loops in the program identified via runtime profiling. The PDG for the hottest loop is constructed over the LLVM IR, with each node representing an instruction in the IR. The memory flow dependences in the PDG that inhibit parallelization are displayed at source level to the programmer, who inserts COMMSET primitives and presents the program back to the compiler. The subsequent compiler passes analyze and transform this program to generate different versions of parallelized code.

4.1 Frontend

The COMMSET parser in the frontend parses and checks the syntax of all COMMSET directives, and synthesizes a C function for every COMMSETPREDICATE. The predicate function computes the value of the C expression specified in the directive. The argument types for the function are automatically inferred by binding the parameters in COMMSETPREDICATE to the COMMSET instances. Type mismatch errors between arguments of different COMMSET instances are also detected. Commutative blocks are checked for enclosing non-local control flow by a top-down traversal of the abstract syntax tree (AST) starting at the node corresponding to the particular commutative block. Finally, global COMMSET meta-data is annotated at the module level, while COMMSET instance data is annotated on individual compound statement or function AST nodes. This meta-data is automatically conveyed to the backend during the lowering phase.

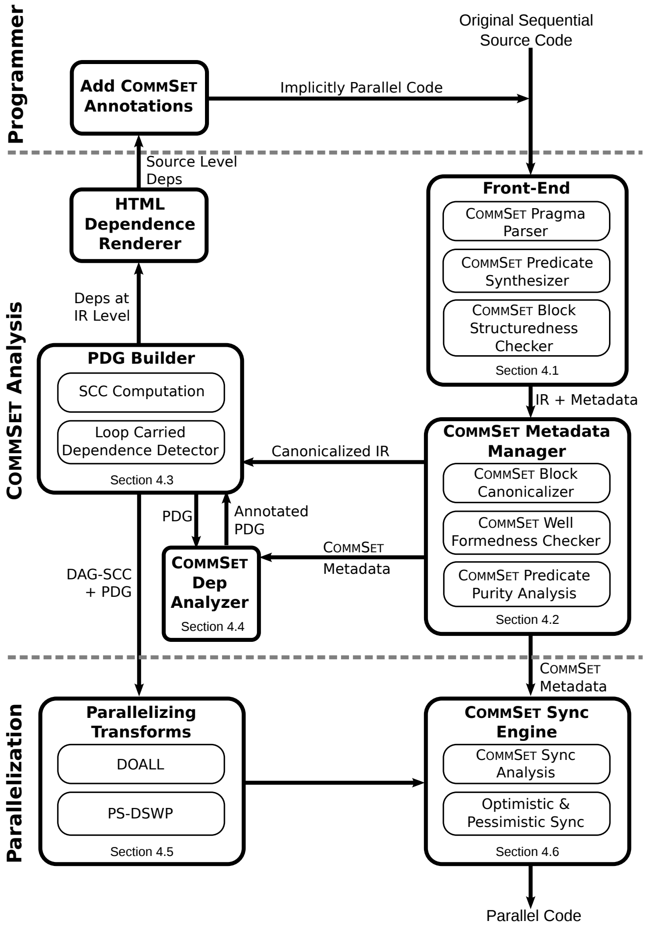


Figure 5: COMMSET Parallelization Workflow

4.2 CommSet Metadata Manager

In the backend, the COMMSET meta-data is an abstraction over the low-level IR constructs, instead of the AST nodes. The COMMSET Metadata Manager processes and maintains a meta-data store for all COMMSET instances and declarations, and answers queries posed by subsequent compiler passes. The first pass of the manager canonicalizes each commutative compound statement, now a structured region (set of basic blocks) within the control flow graph, by extracting the region into its own function. Nested commutative regions are extracted correctly by a post-order traversal on the control flow graph (CFG). The extraction process ensures that arguments specified at a COMMSET instance declaration are parameters to the newly created function. At the end of this pass, all the members of a COMMSET are functions. Call sites enabling optionally commutative named code blocks are inlined to clone the call path from the enabling function call to the COMMSETNAMEDBLOCK declaration. A robust implementation can avoid potential code explosion by automatically extending the interface signature to take in additional arguments for optional commuting blocks. Next, each COMMSET is checked for well-formedness using reachability and cycle detection algorithms on the call graph and the COMMSET graph respectively. The COMMSETPREDICATE functions are tested for purity by inspection of its body.

4.3 PDG Builder

The PDG builder constructs the PDG over the LLVM IR instructions for the target loop using well-known algorithms [12]. A loop carried dependence detector module annotates dependence edges

Algorithm 1: *CommSetDepAnalysis*

```
1  foreach edge $e \in PDG$ do
2      let $n_1 = src(e)$; let $n_2 = dst(e)$;
3      if $typeOf(n_1) \neq Call \lor typeOf(n_2) \neq Call$ then
4          continue
5      end
6      let $Fn(n_1) = f(x_1, \ldots, x_n)$ and $Fn(n_2) = g(y_1, \ldots, y_n)$;;
7      let $S_{in} = CommSets(f) \cap CommSets(g)$;
8      foreach $C_s \in S_{in}$ do
9          if not Predicated($C_s$) then
10             Annotate($e$, $PDG$, $uco$);
11         end
12         else
13             let $f_p = PredicateFn(C_s)$;
14             let $args_1 = CommSetArgs(C_s, f)$;
15             let $args_2 = CommSetArgs(C_s, g)$;
16             let $fargs = FormalArgs(f_p)$;
17             for $i = 0$ to $| args_1 - 1 |$ do
18                 let $x_1 = args_1(i)$ ; let $x_2 = args_2(i)$;
19                 let $y_1 = fargs(2 * i)$ ; let $y_2 = fargs(2 * i + 1)$;
20                 Assert($x_1 = y_1$); Assert($x_2 = y_2$);
21             end
22             if $LoopCarried(e)$ then
23                 Assert($i_1 \neq i_2$); // induction variable;
24                 $r$ = SymInterpret($Body(f_p)$, $true$);
25                 if ($r = true$) and (Dom($n_2$, $n_1$)) then
26                     Annotate($e$, $PDG$, $uco$);
27                 end
28                 else if ($r = true$) then
29                     Annotate($e$, $PDG$, $ico$);
30                 end
31             end
32             else
33                 $r$ = SymInterpret($Body(f_p)$, $true$);
34                 if ($r = true$) then
35                     Annotate($e$, $PDG$, $uco$);
36                 end
37             end
38         end
39     end
40 end
```

as being loop carried whenever the source and/or destination nodes read and update shared memory state.

4.4 CommSet Dependence Analyzer

The COMMSET Dependence Analyzer (Algorithm 1) uses the COMMSET metadata to annotate memory dependence edges as being either unconditionally commutative (uco) or inter-iteration commutative (ico). Figure 2 shows the PDG edges for md5sum annotated with commutativity properties along with the corresponding source annotations. For every memory dependence edge in the PDG, if there exists an unpredicated COMMSET of which both the source and destination's target functions are members, the edge is annotated as uco (Lines 9-11). For a predicated COMMSET, the actual arguments of the target functions at their call sites are bound to corresponding formal parameters of the COMMSETPREDICATE function (Lines 17-19). The body of the predicate function is then symbolically interpreted to prove that it always returns *true*, given the inequality assertions about induction variable values on separate iterations (Lines 21-22). If the interpreter returns true for the current pair of COMMSET instances, the edge is annotated with a commutativity property as follows: A loop carried dependence is annotated as uco if the destination node of the PDG edge dominates the source node in the CFG (Lines 23-34), otherwise it is annotated as ico (Lines 26-27). An intra-iteration dependence edge is always annotated as uco if the predicate is proven to be true (Lines 32-34). Once the commutative annotations are added to the PDG, the PDG builder is invoked again to identify strongly connected components (SCC) [16]. The directed acyclic graph of SCCs (DAG-SCC) thus obtained forms the basis of DSWP family of algorithms [24].

4.5 Parallelizing Transforms

The next step runs the DOALL and PS-DSWP parallelizing transforms, which automatically partition the PDG onto multiple threads for extracting maximal data and pipelined parallelism respectively. For all the parallelizing transforms, the *ico* edges are treated as intra-iteration dependence edges, while *uco* edges are treated as non-existent edges in the PDG. The DOALL transform tests the PDG for absence of inter-iteration dependencies, and statically schedules a set of iterations to run in parallel on multiple threads. The DSWP family of transforms partition the DAG-SCC into a sequence of pipeline stages, using profile data to obtain a balanced pipeline. The DSWP algorithm [25] only generates sequential stages, while the PS-DSWP algorithm [26] can replicate a stage with no loop carried SCCs to run in parallel on multiple threads. Dependences between stages are communicated via lock-free queues in software. Together, the *uco* and *ico* annotations on the PDG enable DOALL, DSWP, and PS-DSWP transforms when previously they were not applicable. Currently, the compiler generates one of each (DSWP, PS-DSWP, and DOALL) schedule whenever applicable, with a corresponding performance estimate. A production quality compiler would typically use heuristics to select the optimal across all parallelization schemes.

4.6 CommSet Synchronization Engine

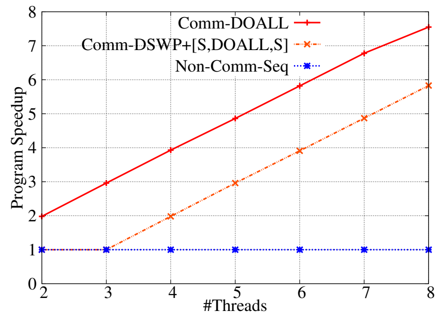
This step automatically inserts synchronization primitives to ensure atomicity of COMMSET members with respect to each other, taking multiple COMMSET memberships into account. The compiler generates a separate parallel version for every synchronization method used. Currently three synchronization modes are supported: optimistic (via Intel's transactional memory (TM) runtime [33]), pessimistic (mutex and spin locks) and lib (well known thread safe libraries or programmer specified synchronization safety for a COMMSET). Initially, the algorithm assigns a unique rank to each COMMSET which determines the global order of lock acquires and releases. The next step determines the set of potential synchronization mechanisms that apply to a COMMSET. Synchronization primitives are inserted for each member of a COMMSET by taking into account the other COMMSETs it is a part of. In the case of TM, a new version of the member wrapped around transactional constructs is generated. For the lock based synchronizations, lock acquires and releases are inserted according to the assigned global rank order. The global ordering along with the acyclic communication primitives that use the lock free queues preserve the invariants required to ensure deadlock freedom [20].

5. Evaluation

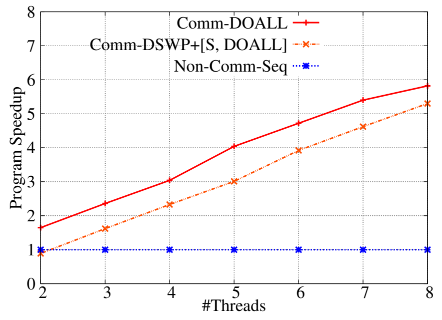
The COMMSET programming model was evaluated on a set of eight programs shown in Table 2. The programs were selected from a repository sourced from a variety of benchmark suites. Seventeen randomly selected programs with potential parallelism-inhibiting memory flow dependencies were examined. Out of these, programs whose hottest loops did not require any semantic changes for parallelization were omitted. For the remaining programs, COMMSET primitives were applied to relax certain execution orders after a careful examination of the intended program semantics. Apart from evaluating parallelization schemes enabled by COMMSET primitives in these programs, alternative parallelization schemes without using COMMSET primitives were also evaluated whenever applicable. Figure 6 shows the speedup of the parallelized programs running on a 1.6GHz Intel Xeon 64-bit dual-socket quad core machine with 8GB RAM running Linux 2.6.24.

Program	Origin	Main loop	Exec. Time	Lines of Code		CommSet Attributes	Parallelizing Transforms	Best Speedup	Best Scheme
				# CommSet Annotations	Total SLOC				
md5sum	Open Src [2]	`main`	100%	10	399	PC, C, S&G	DOALL, PS-DSWP	7.6x	DOALL + Lib
456.hmmer	SPEC2006 [14]	`main_loop_serial`	99%	9	20658	PC, C&I, S&G	DOALL, PS-DSWP	5.8x	DOALL + Spin
geti	MineBench [23]	`FindSomeETIs`	98%	11	889	PI&PC, C&I, S&G	DOALL, PS-DSWP	3.6x	PS-DSWP + Lib
ECLAT	MineBench [23]	`newApriori`	97%	11	3271	PC, C&I, S&G	DOALL, DSWP	7.5x	DOALL + Mutex
em3d	Olden [9]	`initialize_graph`	97%	8	464	I, S&G	DSWP, PS-DSWP	5.8x	PS-DSWP + Lib
potrace	Open Src [29]	`main`	100%	10	8292	PC, C, S&G	DOALL, PS-DSWP	5.5x	DOALL + Lib
kmeans	STAMP [22]	`work`	99%	1	516	C, S	DOALL, PS-DSWP	5.2x	PS-DSWP
url	NetBench [21]	`main`	100%	2	629	I, S	DOALL, PS-DSWP	7.7x	DOALL + Spin

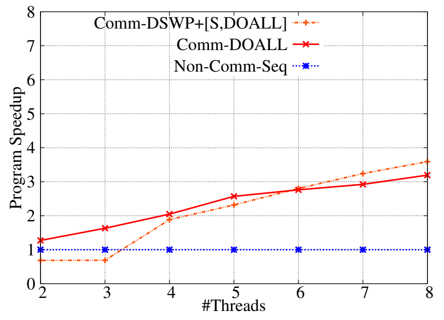
Table 2: Sequential Programs evaluated, their origin, execution time spent in target loop, number of CommSet annotations over sequential code, total number of lines of source code, CommSet features applied (PI: Predication at Interface, PC: Predication at Client, C: Commuting Blocks, I: Interface Commutativity, S: Self Commutativity, G: Group Commutativity), Parallelizing Transforms, Best Speedup Obtained on eight threads, and corresponding Parallelization Scheme with synchronization mechanism (Mutex: Mutex locks, Spin: Spin locks, Lib: Thread-safe Libraries).



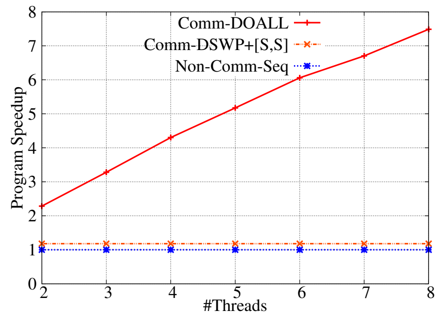
(a) md5sum



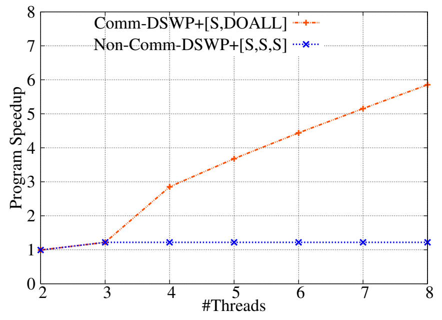
(b) 456.hmmer



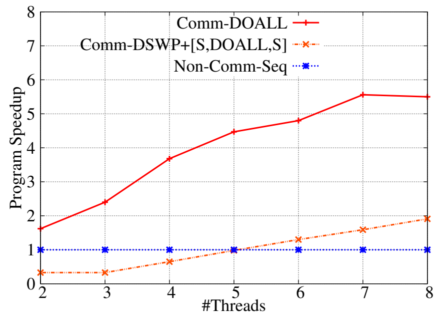
(c) geti



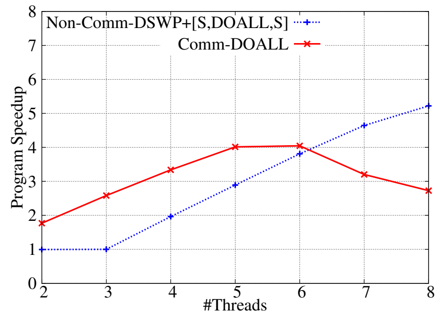
(d) ECLAT



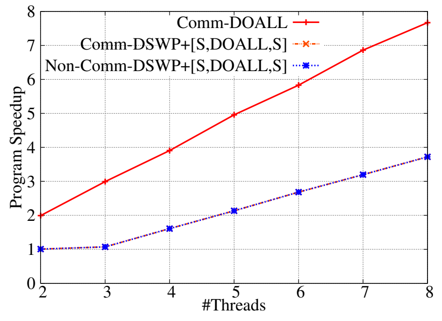
(e) em3d



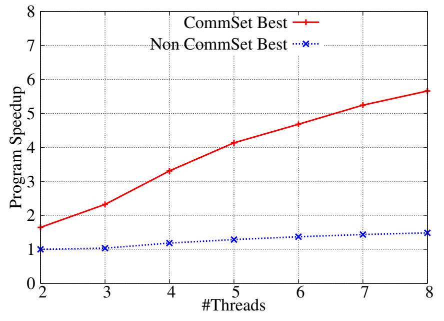
(f) potrace



(g) kmeans



(h) url



(i) Geomean

Figure 6: Performance of DOALL and PS-DSWP schemes using CommSet extensions. Parallelization schemes in each graph's legend are sorted in decreasing order of speedup on eight threads, from top to bottom. The DSWP + [. . .] notation indicates the DSWP technique with stage details within [. . .] (where *S* denotes a sequential stage and *DOALL* denotes a parallel stage). Schemes with *Comm-* prefix were enabled only by the use of CommSet . For each program, the best Non-CommSet parallelization scheme, obtained by ignoring the CommSet extensions is also shown. In some cases, this was sequential execution.

5.1 456.hmmer: Biological Sequence Analysis

456.hmmer performs biosequence analysis using Hidden Markov Models. Every iteration of the main loop generates a new protein sequence via calls to a random number generator (RNG). It then computes a score for the sequence using a dynamically allocated matrix data structure, which is used to update a histogram structure. Finally the matrix is deallocated at the end of the iteration. By applying COMMSET annotations at three sites, all loop carried dependences were broken: (a) The RNG was added to a SELF COMMSET, since any permutation of a random number sequence still preserves the properties of the distribution. (b) The histogram update operation was also marked self commuting, as it performs an abstract SUM operation even though the low-level statements involve floating point additions and subtractions. (c) The matrix allocation and deallocation functions were marked as commuting with themselves on separate iterations. Overall, the DOALL parallelization using spin locks performs best for eight threads, with a program speedup of about 5.82x. A spin lock works better than mutex since it does not suffer from sleep/wakeup overheads in the midst of highly contended operations on the RNG seed variable. The three stage PS-DSWP pipeline, gives a speedup of 5.3x (doing better than the mutex and TM versions of DOALL) by moving the RNG to a sequential stage, off the critical path.

5.2 GETI: Greedy Error Tolerant Itemsets

GETI is a C++ data mining program that determines a set of frequent items that are bought together frequently in customer transactions (*itemsets*). *Itemsets* are implemented as `Bitmap` objects, with items acting as keys. Items are queried and inserted into the `Bitmap` by calls to `SetBit()` and `GetBit()`. Each *itemset* is inserted into an STL vector and then printed to the console. By adding COMMSET annotations at three sites, the main loop was completely parallelizable with DOALL and PS-DSWP: (a) Itemset constructors and destructors are added to a COMMSET and allowed to commute on separate iterations. (b) `SetBit()` and `GetBit()` interfaces were put in a COMMSET predicated on the input key values, to allow for insertions of multiple items to occur out of order. (c) The code block with `vector::push_back()` and prints was context sensitively marked as self commutative in client code. The correctness of this application follows from the set semantics associated with the output. The inter-iteration commutativity properties for constructor/destructor pairs enabled a well performing three-stage PS-DSWP schedule. Transactions were not applicable due to use of external libraries and I/O. Although DOALL schemes initially did better than PS-DSWP, the effects of buffering output indirectly via lock-free queues for PS-DSWP and the increasing number of acquire/release operations for DOALL led to a better performing schedule for PS-DSWP on eight threads. PS-DSWP achieved a limited speedup of 3.6x due to the sequential time taken for console prints but maintained deterministic behavior of the program.

5.3 ECLAT: Association Rule Mining

ECLAT is a C++ program that computes a list of frequent itemsets using a vertical database. The main loop updates objects of two classes `Itemset` and `Lists<Itemset*>`. Both are internally implemented as lists, the former as a client defined class, and the latter as an instantiation of a generic class. Insertions into `Itemset` have to preserve the sequential order, since the `Itemset` intersection code depends on a deterministic prefix. Insertions into the `Lists<Itemset*>` can be done out of order, due to set semantics attached with the output. COMMSET extensions were applied at four sites: (a) Database read calls (that mutate shared file descriptors internally) were marked as self commutative. (b) Insertions into `Lists<Itemset*>` are context-sensitively tagged as self commuting inside the loop. Note that it would be incorrect to tag `Itemset` insertions as self-commuting as it would break the intersection code. (c) Object construction and destruction operations were marked as commuting on separate iterations. (d) Methods belonging to `Stats` class that computes statistics were added to a unpredicated Group COMMSET. A speedup of 7.4x with DOALL was obtained, despite pessimistic synchronization, due to a larger fraction of time spent in computation outside critical sections. Transactions are not applicable due to use of I/O operations. The PS-DSWP transform, using all the COMMSET properties generates a schedule (not shown) similar to DOALL. The next best schedule is from DSWP, that does not leverage COMMSET properties on database read. The resulting DAG-SCC has a single SCC corresponding to the entire inner `for` loop, preventing stage replication.

5.4 em3d: Electro-magnetic Wave propagation

em3d simulates electromagnetic wave propagation using a bipartite graph. The outer loop of the graph construction iterates through a linked list of nodes in a partition, while the inner loop uses a RNG to select a new neighbor for the current node. Allowing the RNG routine to execute out of order enabled PS-DSWP. The program uses a common RNG library, with routines for returning random numbers of different data types, all of which update a shared seed variable. All these routines were added to a common Group COMMSET and also to their own SELF COMMSET . COMMSET specifications to indicate commutativity between the RNG routines required only eight annotations, while specifying pair-wise commutativity would have required 16 annotations. Since the loop does a linked list traversal, DOALL was not applicable. Without commutativity, DSWP extracts a two-stage pipeline at the outer loop level, yielding a speedup of 1.2x. The PS-DSWP scheme enabled by COMMSET directives achieves a speedup of 5.9x on eight threads. A linear speedup was not obtained due to the short execution time of the original instructions in the main loop, which made the overhead of inter-thread communication slightly more pronounced.

5.5 potrace: Bitmap tracing

potrace vectorizes a set of bitmaps into smooth, scalable images. The code pattern is similar to md5sum, with an additional option of writing multiple output images into a single file. In the code section with the option enabled, the SELF COMMSET annotation was omitted on file output calls to ensure sequential output semantics. The DOALL parallelization yielded a speedup of 5.5x, peaking at 7 threads, after which I/O costs dominate the runtime. For the PS-DSWP parallelization, the sequentiality of image writes limited speedup to 2.2x on eight threads.

5.6 kmeans: K means clustering algorithm

kmeans clusters high dimensional objects into similar featured groups. The main loop computes the nearest cluster center for each object and updates the center's features using the current object. The updates to a cluster center can be re-ordered, with each such order resulting in a different but valid cluster assignment. Adding the code block that performs the update to a SELF COMMSET breaks the only loop carried dependence in the loop. The DOALL scheme with pessimistic synchronization showed promising speedup until five threads (4x), beyond which frequent cache misses due to failed lock/unlock operations resulted in performance degradation. Transactions (not shown) do not help either, with speedup limited to 2.7x on eight threads. The three-stage PS-DSWP scheme was best performing beyond six threads, showing an almost linear performance increase by executing the cluster update operation in a third sequential stage. It achieved a speedup of 5.2x on eight threads. This highlights the performance gains achieved by moving highly contended dependence cycles onto a sequential stage, an important insight behind the DSWP family of transforms.

5.7 URL: url based switching

The main loop in the program switches a set of incoming packets based on its URL and logs some of the packet's fields into a file. The underlying protocol semantics allows out-of-order packet switching. Adding the function to dequeue a packet from the packet pool and the logging function to SELF COMMSETs broke all the loop carried flow dependences. No synchronization was necessary for the logging function while locks were automatically inserted to synchronize multiple calls to the packet dequeuing function. A two stage PS-DSWP pipeline was also formed by ignoring the SELF COMMSET annotation on the packet dequeue function. The DOALL parallelization (7.7x speedup on eight threads) outperforms the PS-DSWP version (3.7x on eight threads) because of low lock contention on the dequeue function and the overlapped parallel execution of the packet matching computation.

5.8 Discussion

The application of COMMSET achieved a geomean speedup of 5.7x on eight threads for the programs listed in Table 2, while the geomean speedup for Non-COMMSET parallelizations is 1.49x (Figure 6i). For four out of the eight programs, the main loop was not parallelizable at all without the use of COMMSET primitives. With the application of COMMSET, DOALL parallelization performs better than PS-DSWP on 5 benchmarks, although PS-DSWP has the advantage of preserving deterministic output in two of them. For two of the remaining programs, PS-DSWP yields better speedup since its sequential last stage performs better than concurrently executing COMMSET blocks in the high lock contention scenarios. DOALL was not applicable for em3d, due to pointer chasing code. In terms of programmer effort, an average of 8 lines of COMMSET annotations were added to each program to enable the various parallelization schemes. Predication based on the client state, as a function of the induction variable enabled well performing parallelizations without the need for runtime checks. The use of commuting blocks avoided the need for major code refactoring. The applicability of COMMSET compared favorably to the applicability of other compiler based techniques like Paralax and VELOCITY. VELOCITY and Paralax cannot be used to parallelize four benchmarks: geti, eclat, md5sum and potrace since they do not support predicated commutativity. For 456.hmmer, VELOCITY would require a modification of 45 lines of code (addition of 43 lines and removal of 2 lines) in addition to the commutativity annotations. COMMSET did not require those changes due to the use of named commuting blocks.

6. Related Work

Semantic Commutativity based Parallelizing Systems. Jade [27] supports object-level commuting assertions to specify commutativity between every pair of operations on an object. Additionally, Jade exploits programmer written read/write specifications for exploiting task and pipeline parallelism. The COMMSET solution relies on static analysis to avoid read/write specifications and runtime profiles to select loops for parallelization. Galois [18], a runtime system for optimistic parallelism, leverages commutativity assertions on method interfaces. It requires programmers to use special set abstractions with non-standard semantics to enable data parallelism. The COMMSET compiler currently does not implement runtime checking of COMMSETPREDICATEs required for optimistic parallelism. However, the COMMSET model is able to extract both data and pipelined parallelism without requiring any additional programming extensions. DPJ [5], an explicitly parallel extension of Java uses commutativity annotations at function interfaces to override restrictions placed by the type and effect system of Java. Several researchers have also applied commutativity properties for semantic concurrency control in explicitly parallel settings [10, 17]. Paralax [30] and VELOCITY [6, 7] exploit self-commutativity at the interface level to enable pipelined parallelization. VELOCITY also provides special semantics for commutativity between pairs of memory allocation routines for use in speculative parallelization. Compared to these approaches, the COMMSET language extension provides richer commutativity expressions. Extending the compiler with a speculative system to support all COMMSET features at runtime is part of future work. Table 1 summarizes the relationship between COMMSET and the above programming models.

Implicit Parallel Programming. OpenMP [3] extensions require programmers to explicitly specify parallelization strategy and concurrency control using additional primitives ("`#pragma omp for`", "`#pragma omp task`", "`critical`", "`barrier`", etc.). In the COMMSET model, the choice of parallelization strategy and concurrency control is left to the compiler. This not only frees the programmer from having to worry about low-level parallelization details, but also promotes performance portability. Implicit parallelism in functional languages have been studied recently [13]. IPOT [31] exploits semantic annotations on data to enable parallelization. Hwu et al. [15] also propose annotations on data to enable implicit parallelism. COMMSET extensions are applied to code rather than data, and some annotations like `reduction` proposed in IPOT can be easily integrated with COMMSET.

Compiler Parallelization. Research on parallelizing FORTRAN loops with regular memory accesses [11, 16] in the past has been complemented by more recent work on irregular programs. The container-aware [32] compiler transformations parallelize loops with repeated patterns of collections usage. Existing versions of these transforms preserve sequential semantics. The COMMSET compiler can be extended to support these and other parallelizing transforms without changes to the language extension.

Commutativity Analysis. Rinard et al. [28] proposed a static analysis that determines if commuting two method calls preserves concrete memory state. Aleen et al. [1] apply random interpretation to probabilistically determine function calls that commute subject to preservation of sequential I/O semantics. Programmer written commutativity assertions are more general since they allow multiple legal outcomes and give a programmer more flexibility to express intended semantics within a sequential setting.

7. Conclusion

This paper presented an implicit parallel programming solution based on a unified, syntactically succinct and generalized semantic commutativity construct called COMMSET. The model provides programmers the flexibility to specify commutativity relations between arbitrary structured blocks of code and does not require the use of any additional parallel constructs. Parallelism exposed implicitly using COMMSET is independent of any particular parallelization strategy or concurrency control mechanism. A complete end-to-end implementation of COMMSET exists in a parallelizing compiler. Evaluation of eight real world programs indicates that the use of COMMSET extensions and the associated compiler technology enables scalable parallelization of programs hitherto not amenable to automatic parallelization. This demonstrates the effectiveness of the COMMSET construct in exposing different parallelization opportunities to the compiler.

Acknowledgments

We thank the entire Liberty Research Group for their support and feedback during this work. We also thank the anonymous reviewers for their insightful comments. This material is based on work supported by National Science Foundation Grants 1047879, 0964328, and 0627650, and United States Air Force Contract FA8650-09-C-7918.

References

[1] F. Aleen and N. Clark. Commutativity analysis for software parallelization: Letting program transformations see the big picture. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[2] Apple Open Source. md5sum: Message Digest 5 computation. `http://www.opensource.apple.com/darwinsource/`.

[3] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 2009.

[4] G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of NESL. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 1996.

[5] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for Deterministic Parallel Java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2009.

[6] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multicore. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007.

[7] M. J. Bridges. *The VELOCITY compiler: Extracting efficient multicore execution from legacy sequential codes*. PhD thesis, 2008.

[8] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[9] M. C. Carlisle. *Olden: Parallelizing programs with dynamic data structures on distributed-memory machines*. PhD thesis, 1996.

[10] B. D. Carlstrom, A. McDonald, M. Carbin, C. Kozyrakis, and K. Olukotun. Transactional collection classes. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2007.

[11] R. Eigenmann, J. Hoeflinger, Z. Li, and D. A. Padua. Experience in the automatic parallelization of four Perfect-benchmark programs. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing (LCPC), 1992*.

[12] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3), 1987.

[13] T. Harris and S. Singh. Feedback directed implicit parallelism. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2007.

[14] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 2006.

[15] W.-m. Hwu, S. Ryoo, S.-Z. Ueng, J. Kelm, I. Gelado, S. Stone, R. Kidd, S. Baghsorkhi, A. Mahesri, S. Tsao, N. Navarro, S. Lumetta, M. Frank, and S. Patel. Implicitly parallel programming models for thousand-core microprocessors. In *Proceedings of the 44th annual Design Automation Conference (DAC)*, 2007.

[16] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: a Dependence-based Approach*. Morgan Kaufmann Publishers Inc., 2002.

[17] E. Koskinen, M. Parkinson, and M. Herlihy. Coarse-grained transactions. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2010*.

[18] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[19] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of 2nd International Symposium on Code Generation and Optimization (CGO)*, 2004.

[20] R. Leino, P. Müller, and J. Smans. Deadlock-free channels and locks. In *Proceedings of the 19th European Symposium on Programming (ESOP)*, 2010.

[21] G. Memik, W. H. Mangione-Smith, and W. Hu. NetBench: a benchmarking suite for network processors. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2001.

[22] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2008.

[23] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. MineBench: A benchmark suite for data mining workloads. In *IEEE International Symposium on Workload Characterization (IIWSC)*, 2006.

[24] G. Ottoni. *Global Instruction Scheduling for Multi-Threaded Architectures*. PhD thesis, 2008.

[25] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2005.

[26] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2008.

[27] M. C. Rinard. *The design, implementation and evaluation of Jade, a portable, implicitly parallel programming language*. PhD thesis, 1994.

[28] M. C. Rinard and P. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI)*.

[29] P. Selinger. potrace: Transforming bitmaps into vector graphics. `http://potrace.sourceforge.net`.

[30] H. Vandierendonck, S. Rul, and K. De Bosschere. The Paralax infrastructure: Automatic parallelization with a helping hand. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[31] C. von Praun, L. Ceze, and C. Caşcaval. Implicit parallelism with ordered transactions. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2007.

[32] P. Wu and D. A. Padua. Beyond arrays - a container-centric approach for parallelization of real-world symbolic applications. In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing (LCPC), 1999*.

[33] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. S. Lee. Kicking the tires of software transactional memory: Why the going gets tough. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2008.

[34] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *Proceedings of 14th International Conference on High-Performance Computer Architecture (HPCA)*, 2008.