Enabling Programmer-controlled Combined Memory Consistency for Compiler Optimization

Dixie M. Hisley
U.S. Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5067
hisley@arl.mil

Lori L. Pollock
Computer and Information Sciences
University of Delaware
Newark, DE 19716
pollock@cis.udel.edu

ABSTRACT

In this paper, we present a novel compiler technique that provides the capability to guarantee a different memory consistency model for different variables within the same shared memory parallel program. Programmers need only focus on their parallel algorithm and which variables can tolerate the use of old values, while the compiler automatically guarantees sequential consistency for all remaining variables and identifies where unnecessary synchronization overhead can be avoided. Program slicing is exploited to address the interactions between variables that can tolerate old values and those that require the most recent value to be read.

KEYWORDS: compilers, shared memory parallel programming, memory consistency models

1. INTRODUCTION

Despite the purported cost effectiveness of workstation clusters, scientists continue to run computationally intensive applications on high-end shared memory multiprocessor architectures. While scientists convert their programs to explicitly parallel programs, they rely on the compiler to identify and exploit performance gains for each processor and also to maintain a "correct" program execution similar to that of a uniprocessor undertaking concurrent execution of multiple tasks. Unfortunately, potential performance gains can be sacrificed, with much of the degradation due to the memory consistency model and the compiler's interactions with the hardware memory consistency model. Although optimized sequential code can outperform unoptimized compiled code by as much as the performance difference between two generations of processor hardware [13], classical compiler analysis and optimizations [1, 11] were developed in the context of sequential programs designed to run on uniprocessors. These techniques do not account for updates to shared variables in threads other than the thread being analyzed. Thus, these compiler optimizations are typically turned off in order to avoid generating a program that does not behave as the user expects, given the underlying hardware memory consistency model.

In this paper, we present a novel compiler technique that provides the capability to guarantee a different memory consistency model for different variables within the same shared memory parallel program. The key insight is that programmers are cognizant of which variables can tolerate old values and which variables need to be accessed in a way that all processors always see the most recently stored value. We refer to the first set of variables as *nonsecure* and the second set as *secure*.

In our approach, programmers add *secure*, *nonsecure* annotations to shared variable declarations to indicate their expectations, which are automatically propagated to the statements manipulating shared variables. We construct a *relaxed* concurrent static single assignment form of the program [8], so existing compiler optimization techniques for explicitly parallel, shared memory programs can be applied without modification. Sequential consistency is enforced selectively by the placement of fence instructions to ensure that the most recent value of a secure variable is always read. In other cases, we default to location consistency (LC), which is based on the partial order execution semantics of parallel programs [3]. The selective enforcement of sequential or location consistency is complicated by the interaction between the manipulation of secure and nonsecure variables throughout the program. We use program slicing to identify and address these interactions.

To our knowledge, this is the first compiler technique to enable multiple memory consistency models to be applied within the same program. Previous research advocated the use of a single model for a given compiler/architecture combination. We believe that parallel programmers should have the option to specify whether they want the semantics of a particular consistency model for different parts of a program based on their knowledge of their algorithm and data structures. The benefit is relieving the programmer of understanding memory consistency models, limiting the overhead of sequential consistency to those parts of the program related to the programmer's expectations of sequential consistency, and tailoring memory consistency modeling to individual programs automatically.

The remainder of the paper is organized as follows. Section 2 provides background on characteristics of applications, shared memory architectures and memory consistency issues. Section 3 discusses the uniprocessor optimization problem within a parallel programming environment and overviews previous work on this problem. Section 4 presents our technique for a programmer-controlled memory consistency model. Section 5 summarizes our conclusions and future work.

2. PARALLEL PROGRAMMING AND MEMORY CONSISTENCY

This research focuses on the loop-level and single program, multiple data (SPMD) shared memory parallel programming specifically targeted by OpenMP. For shared memory programs, eliminating races on shared data accesses is necessary to produce correct repeatable results. However, some algorithms rely on intentional data races as a natural part of their algorithm. An example is iterative algorithms based on chaotic relaxation such as the parallel successive-over-relaxation (SOR) method.

Nondeterministic parallel programs with data races are the most challenging for compiler analysis and optimization. However, one approach is to use a weaker memory consistency model, like location consistency. This weaker model does not consider data races to be errors, but instead gives them different semantics that allow the same flexibility in reordering data accesses as for parallel programs that are deterministic or nondeterministic data-race-free. In LC, instead of assuming that all writes to the same memory location are serialized according to some total order, the state of a memory location is modeled as a partially ordered multiset (pomset) of write and synchronization operations. There is no requirement for all processors to observe the same ordering of concurrent write operations; only the partial order (the order of writes on a single processor with respect to that processor) defined by the program must be preserved. As a result, the LC model provides a simple contract between the programmer (or compiler) and the hardware. Our work is motivated by the fact that very large applications can have both secure and nonsecure variables used in ways that whole subroutines could obtain higher performance from the LC model while other parts of the program are guaranteed to abide by the sequential consistency model to reflect programmer expectations.

3. UNIPROCESSOR OPTIMIZATION OF SHARED MEMORY PROGRAMS

Midkiff and Padua [9] demonstrated that straightforward application of sequential optimization techniques within compilers for explicitly parallel shared memory programming fail to maintain correctness. Data races and synchronization issues make it impossible to apply classical methods directly to explicitly parallel programs. In order to incorporate uniprocessor optimizations into a compiler for explicitly parallel programs, one must design an appropriate intermediate program representation and algorithms for analyzing when to apply the various desired uniprocessor optimizations in the parallel program.

Some of the earliest papers on the analysis and optimization of explicitly parallel programs on shared memory computers explored the minimal set of delays that enforce sequentially consistent execution of concurrent processes [6, 10, 15]. In order to extend optimizations to work correctly in a parallel programming environment, concurrent forms of the control flow graph and the static single assignment framework were proposed by previous researchers [2, 7, 13, 14]. In particular, intermediate representations for parallel programs include the parallel program graph [13], the parallel flow graph [2], and the parallel precedence graph with static single assignment form [14]. Lee [7], and Lee and Padua [8] developed a concurrent static single assignment form (CSSA) based on the concurrent control flow graph (CCFG). Novillo [12] extended the concurrent control flow graph and the CSSA form for programs with lock/unlock synchronization. Knoop and Steffen [5] introduced efficient and optimal bitvector analyses for parallel programs using the control flow graph.

Overall, previous research for explicitly parallel codes has focused on developing correctness criteria, intermediate program representations, and extending and applying classical data flow analysis and optimization techniques. However, much of the analysis is restricted to a subset of parallel programs and does not deal with all types of explicit synchronization. In addition, there has been a lack of consensus on an acceptable memory consistency model for developing correctness criteria [3]. Very few realistic implementations of the analysis and optimizations have been constructed.

4. PROGRAMMER-CONTROLLED MEMORY CONSISTENCY

Our overall goal is to give the programmer control and flexibility in memory consistency modeling via simple program annotations and to develop automatic techniques that translate the programmer's requests into modifications to the intermediate program representation in such a way that optimization and analysis algorithms require few changes. In this paper, we present an approach to programmer-controlled memory consistency that is based on the data flow of the program. Shared variables that need to satisfy the memory coherence assumption are annotated by the programmer as *secure*. Shared variables that do not need to satisfy the memory coherence assumption should be annotated as *nonsecure*. The annotations are placed on the shared variable declaration statements indicated by the

!$OMP SHARED construct (see Figure 2b). The scope of the annotation is the same as the scope of the shared variable, that is, the parallel region associated with the parallel directive.

4.1 Definitions

Sequential consistency has been historically defined in terms of the total program order for a sequential program or partial orderings for parallel programs. For this research, we need to extend the definition of sequential consistency for parallel programs to be defined in terms of a single variable with respect to a given point in a program. The definitions of sequentially consistent, secure, and nonsecure shared variable are given as follows:

Definition 4.1 Sequentially Consistent Shared Variable. *A* **sequentially consistent shared variable with respect to a given point** *P in a program requires that all updates to the shared variable that could affect the value of the variable at point P be immediately visible to other threads (i.e., abide by the memory coherence assumption).*

Definition 4.2 Secure Variable. *A* **secure variable** *is a shared variable that is referenced by different threads that might execute concurrently in an explicitly parallel program and needs to adhere to the memory coherence assumption at all program points P and is therefore accessed according to the constraints imposed by sequential consistency.*

Definition 4.3 Nonsecure Variable. *A* **nonsecure variable** *is a shared variable that is referenced by different threads that might execute concurrently in an explicitly parallel program and does not need to adhere to the memory coherence assumption and therefore can be accessed according to the constraint imposed by location consistency (i.e., that each processor sequentially executes its node program).*

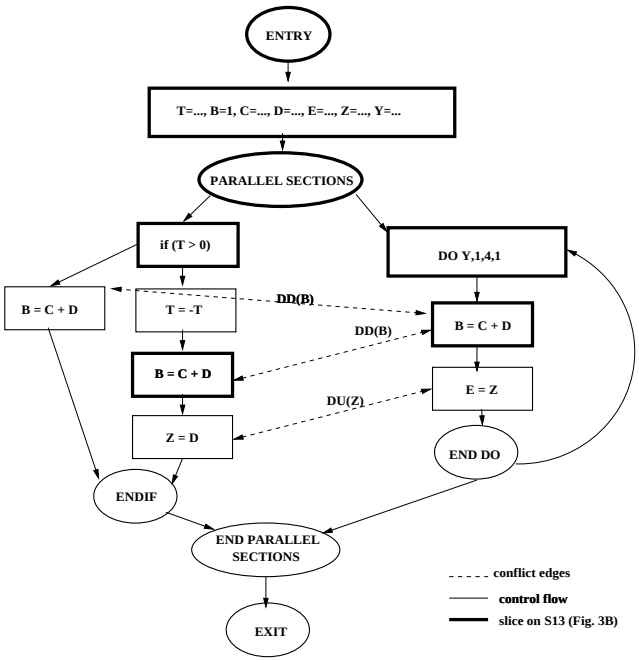

Figure 1. Example CCFG with Conflict Edges

Since the LC memory consistency model is used as the default model, *secure* variables that are involved in data races must be made to satisfy the memory coherence assumption. This can be accomplished by inserting constructs (by the programmer or by the compiler) to synchronize shared variable references or by the compiler automatically switching to a sequential consistency memory model to guarantee correctness. In this research, we take the approach of automatically switching to a sequential consistency model within the compiler. Therefore, all shared variables that are annotated to be *secure* will be guaranteed to be sequentially consistent shared variables. All shared variables that are annotated to be *nonsecure* can default to location consistency whenever possible.

4.2 Base Intermediate Representation

In particular, our approach is based on the concurrent control flow graph (CCFG) program representation [7, 8]. The CCFG is a directed graph G = ⟨N, E, $Entry_G$, $Exit_G$⟩ such that N is the set of nodes in the graph (with each node corresponding to a basic block), E is the set of control flow edges, conflict edges, and synchronization edges, and $Entry_G$, $Exit_G$ are the unique entry and exit points of the program. A conflict edge is a bidirectional edge in the CCFG that joins any two basic blocks that can be executed concurrently and reference the same shared variable (one of

```
S1:  T=…
S5:  E=…
S6:  Z=…
S7:  Y=…
S2:  B=…
S3:  C=…
S4:  D=…
S8:  !$OMP PARALLEL SECTIONS
S9:  !$OMP SHARED              !$OMP SHARED (T,B,C,Y) SECURE
       (T,B,C,D,E, Z,Y)        !$OMP SHARED (D,E,Z) NONSECURE
S10: IF (T > 0) THEN
S11:   T = -T
S12:   B = C+ D
S13:   Z = D
S14: ELSE
S15:   B = C + D
S16: ENDIF
S17: !$OMP SECTION
S18: DO Y = 1,4
S19:   B = C + D
S20:   C = Z
S21: ENDO
S22: !$OMP END PARALLEL SECTIONS
```

(a) **(b)**

Figure 2. (a) Example OpenMP (b) Annotated Stmts

the references is a write reference). There are two kinds of conflict edges: def-use and def-def.

A separate CCFG is generated for each procedure. Thus, a basic form of interprocedural analysis information can be gathered. At each procedure call, shared variables referenced and mutex bodies defined by the called procedure are propagated to the call site. This allows conflict and synchronization analysis to treat function calls almost as if they were inlined [7]. Figure 1 presents an example CCFG with conflict edges for the sample code shown in Figure 2(a). The solid edges in the CCFG are control flow edges, while conflict edges are represented by dashed edges. For example, there is a def-def edge linking the two assignments to variable B in the two different threads. A def-use edge for variable Z indicates the relationship between the two threads created by the statements involving variable Z.

We begin with a CCFG representation of a program, which includes all conflict edges that would have to be enforced, assuming a sequential consistency model for the whole program. Our compilation phase produces a relaxed CCFG in which conflict edges have been removed according to the programmer's annotations on shared variables.

4.3 Modeling Interactions of Variables

In order to ensure that all reads of a given secure variable indeed see the most recent value according to sequential consistency, we need to ensure that sequential consistency is enforced for any value on which these reads depend, either directly or indirectly. In terms of the CCFG, the programmer is presented with sequential consistency, and the underlying relaxed consistency model of the hardware is hidden by inserting def-def and def-use conflict edges for each shared variable def-def or def-use dependency between different threads and by inserting fence instructions to ensure that the conflict edges are enforced.

In general, both def-def and def-use conflict edges for shared variables annotated as nonsecure could be removed and still maintain sequential consistency for all secure variables, as long as nonsecure variables were manipulated completely in isolation of secure variables. However, programmers create dependencies among secure and nonsecure variables, which prevents the compiler from blindly removing all conflict edges for nonsecure variables.

The dependencies among reads and writes of secure and nonsecure variables can be specified in terms of four relations:

(1) S: $LHS_{secure} = f(RHS_0, RHS_1,...)$, where LHS_{secure} is the definition of a secure variable as a function of a set of variables.
   (a) Each secure RHS variable in S represents a use of a secure variable within a statement defining another secure variable, LHS_{secure}. Any conflict edges leading into S due to these uses must be maintained for two reasons: (1) to ensure that the read of the RHS variable in S is the most recent value and (2) to ensure that the def of the secure variable in the LHS also maintains sequential consistency for later reads of LHS_{secure}.
   (b) Each nonsecure RHS variable in S represents a use of a nonsecure variable within a statement defining a secure variable. This case represents one way in which the manipulation of secure and nonsecure variables can interact. We discuss this case in more depth below.

(2) T: $LHS_{nonsecure} = f(RHS_0, RHS_1,...)$, where $LHS_{nonsecure}$ is the definition of a nonsecure variable as a function of a set of variables.
   (a) Each nonsecure RHS variable in T represents a use of a nonsecure variable $LHS_{nonsecure}$. Because we default to the LC model for nonsecure variables, any conflict edges leading into T due to the nonsecure RHS variable can be removed as long as the value of $LHS_{nonsecure}$ is not used later in a computation defining a secure variable. The def of the nonsecure RHS variable in this statement need not be performed prior to this read of the RHS variable.
   (b) Each secure RHS variable in T represents a use of a secure variable within a statement defining a nonsecure variable. This is the second case where secure and nonsecure variable manipulations interact.

In case (1b), sequential consistency for LHS_{secure} implies that for all variable uses, RHS_i (in the computation of LHS_{secure}), all conflict edges leading to statement S for RHS_i must be maintained in order to ensure that LHS_{secure} is kept secure for future reads. Thus, any RHS_i that is nonsecure, call it $RHS_{nonsecure,}$ must be treated as a secure variable for this particular use at S. Due to the requirement to treat $RHS_{nonsecure}$ as secure at this statement, conflict edges leading into S for $RHS_{nonsecure}$ must be maintained, and furthermore, any reaching def of $RHS_{nonsecure}$ at S must also be analyzed for dependencies on nonsecure variables.

For case (2b), since the LHS is nonsecure, location consistency will be used for its def, and thus, typically any conflict edges leading into this statement for any of its RHS variables (which this statement reads) can be safely removed to reflect the more relaxed consistency model. However, the programmer has indicated that the most recent value of secure variables be read. A secure RHS variable in T would correspond to one of these reads. Thus, the conflict edges leading into T for any secure variables in the

RHS must remain, regardless of the nonsecure status of the LHS.

The programmer annotated the shared variable declarations, but the dependencies between secure and nonsecure variables suggests that the compiler needs to determine additional locations where nonsecure variables need to be treated like secure variables in order to determine which conflict edges can be removed. This does not imply that all occurrences of the nonsecure variable need to be treated like a secure variable, only those on which a secure variable depends, either directly or indirectly.

The key insight is that the relevant dependencies that need to be uncovered in order to implement different consistency models in different parts of a program based on data flow can be identified by program slicing [16]. A program slice detects the parts of a program that statically may affect the values computed at some point. A slice on a particular variable at a given program point will reveal the dependencies between secure and nonsecure variables on which this variable's value depends. For example, by slicing on a RHS variable at statement S, we can relabel its use at S and the variable defs and uses at statements that directly or indirectly affect it to be marked as secure when the LHS variable must be secure at S. Thus, for case (1a), we need to slice on each nonsecure RHS, relabeling the RHS use as well as any variable manipulations that potentially affect it to be secure.

An algorithm for the static slicing of threaded programs based on control flow graphs and program dependence graphs has been developed by Krinke [4]. This algorithm is leveraged for use in our work. The input required for this slicing algorithm is a threaded program dependence graph (tPDG) and the slicing criterion (i.e., a node of the tPDG and the variable(s) to slice on). The output of the algorithm is the slice, which consists of a set of nodes of the tPDG. The slicing algorithm is referred to as function *parallel_slice* in Figure 3, which presents the pseudocode for our algorithm to implement programmer-controlled memory consistency.

4.4 Algorithm for Building Relaxed CCFG

There are three phases in the construction of a relaxed CCFG. First, an initialization phase is performed to build a sequentially consistent CCFG in CSSA form (which includes all conflict edges as if all variables are secure), extended to include control and data dependence edges. Our algorithm takes this extended CCFG representation as input. In our algorithm, we assume that each node of the CCFG is an individual statement. Because nonsecure variables can be determined to require secure treatment at selected statements, the secure and nonsecure annotations on OpenMP shared variable declarations are propagated as flags on every def and use of the shared variables in the CCFG. In this way, programmers need only add annotations on variable declarations, but their expectations are propagated automatically to all defs and uses of the shared variables.

In the second phase, backward slicing is performed on each case (1b) in the CCFG. During the backward slicing, any nonsecure variable reference found in the slice, including the original RHS variable, are relabeled to be secure. No conflict edges are removed during this phase. Finally, the third phase eliminates conflict edges by a single pass over the CCFG using the final secure and nonsecure flags to identify all case (2a) instances. Figure 3 presents pseudocode for our algorithm.

Algorithm Construct_Relaxed_CCFG.

Input: Sequentially consistent concurrent control flow graph G = ⟨N, E, $Entry_G$, $Exit_G$ ⟩ that includes all conflict edges as if all variables are secure in CSSA form, with added control and data dependence edges for slicing.

Output: A CCFG in CSSA form with conflict edges eliminated for nonsecure variables that can be handled using location consistency.

```
1: // Initialize flags on every def or use of shared variables
2: foreach shared variable S_i do
3: Initialize annotate_flag to secure or nonsecure to reflect annotation
4: end for
5: foreach shared variable S_i in node n_i ∈ N do
6: label_flag = annotate_flag
7: end for
8: // Perform backwards slicing to relabel nonsecure uses and defs*/
9:  foreach node n_i ∈ N with LHS_i label_flag = secure do
10:         foreach RHS_j with label_flag = nonsecure do
11:        // return slice w, a worklist containing nodes that affect
RHS_i
12:           worklist w = parallel_slice (RHS_i, n_i,)
13:           foreach node n_j ∈ w do
14:              foreach shared variable S_i in node n_j do
15:                    label_flag = secure
16:              end for
17:           end for
18:        endfor
19: endfor
20: /* Test for case 2a and eliminate conflict edges
21:  foreach node n_i ∈ N do
22:         if LHS S_i label_flag = nonsecure then
23:         delete all def-def conflict edges to LHS variable of S_i
24:         delete all def-use conflict edges to nonsecure RHS
variables of S_i
25:        endif
26: endfor
```

Figure 3. Algorithm for Constructing Relaxed CCFG

In Figure 1, we assume that the variables T, B, C, and Y are initially annotated by the programmer as secure, while variables D, E, and Z are annotated as nonsecure (as indicated in Figure 2b). The implications of these annotations are the removal of the def-use (Z) conflict edge and the relabeling of variable D to be secure in the two statements with C + D expressions because B is a secure variable being defined at those statements (case 1b). The algorithm for programmer-controlled memory consistency

would slice from the statements B = C + D on variable D and relabel all nonsecure references in the program slice for D to become secure, in response to the secure annotation on B. Since variable Z is annotated to be nonsecure, variable Z in the statement Z = D can be location consistent, and the conflict edge to this LHS variable can be eliminated (case 2b).

The CCFG is built using a slightly modified version of a standard algorithm to build control flow graphs [1]. The CSSAME form of the CCFG can be built in $O(r^3)$ time, where r is the maximum of the number of nodes, number of control edges, number of assignments, and number of variable references in the program [12]. The slicing algorithm takes the same time and space complexity as unthreaded slicing, with additional time complexity that could be exponential in the number of conflict edges in the worst case; however, much more reasonable times are expected for real programs given the characteristics of conflict edges we have observed and Krinke has observed [4]. Slicing is performed once for each variable use labeled nonsecure in each statement with a LHS variable that is originally labeled as secure. Slicing is not repeatedly applied as relabeling occurs. The pass that eliminates the conflict edges requires O(N) time, where N is the number of nodes in the CCFG.

5. CONCLUSIONS AND FUTURE WORK

The major contribution of this paper is a technique for programmer-controlled memory consistency for explicitly parallel programs in which different models of consistency prevail in different parts of the program. Sequential consistency overhead is limited to where programmers want to ensure most recent values are seen, while programmers are relieved of having to understand memory consistency models.

Compiler optimizations can be tailored to different memory consistency models for different programs and for different parts of the program. We are designing a software analysis tool based on this approach to perform program analysis and optimization for real scientific OpenMP programs. An existing research compiler infrastructure, Odyssey [12], is being extended to incorporate the technique described in this paper. We have gathered a set of scientific OpenMP benchmarks. *irreg* simulates an unstructured computational fluid dynamics code, *jacobi* uses a successive-over-relaxation iterative algorithm to discretize the Helmholtz equation, and *md* simulates a molecular dynamics program. Experimentation will be performed to determine improved performance in terms of cost, precision, and/or scalability.

The approach to programmer-contirolled memory consistency presented in this paper is based on annotating shared variables and utilizing data flow to partition the program's consistency modeling. We plan to investigate alternative ways of providing programmer-controlled memory consistency with the goal of allowing multiple models to be in effect during optimization based on the program and programmer's expectations.

6. REFERENCES

[1] A. Aho, R. Sethi, & J. Ullman, *Compilers: Principles, Techniques, and Tools* (NY: Addison Wesley, 1986).

[2] D. Grunwald & H. Srinivasan, Data flow equations for explicitly parallel programs, *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Progr. (PPoPP'93)*, *28*(7), ACM SIGPLAN Not., 1993, 159-168.

[3] G. Gao & V. Sarkar, Location consistency: stepping beyond the barriers of memory coherence and serializability, *ACAPS Technical Memo 78*, School of Computer Science, McGill University, Montreal, Quebec, December 1994.

[4] J. Krinke, Static slicing of threaded programs, *Proc. of ACM SIGPLAN Workshop on Program Analysis for Software Tools and Engineering*, Montreal, Canada, June 1998.

[5] J. Knoop & B. Steffen, Code motion for explicitly parallel programs, *Proc. ACM SIGPLAN Symp. On Principles and Practice of Parallel Progr. (PPoPP '99)*, 1999,13-24.

[6] A. Krishnamurthy & K. Yelick, Analyses and optimizations for shared address space programs, *Parallel and Distributed Computing, 38*, 1996, 139-144.

[7] J. Lee, *Compilation Techniques for Explicitly Parallel Programs*, Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1999.

[8] J. Lee & D. Padua, Hiding relaxed memory consistency with compilers, *Proc. of International Conference on Parallel Architectures and Compilation Techniques*, 2000, 111-122.

[9] S. Midkiff & D. Padua, Issues in the optimization of parallel programs, *Proc. Int. Conf. On Parallel Processing, II*, 1990, 105-113.

[10] S. Midkiff, D. Padua, & R. Cytron, Compiling programs with user parallelism, *Languages and Compilers for Parallel Computing*, 1990, 402-422.

[11] S. Muchnick, *Advanced Compiler Design and Implementation (CA:* Morgan Kaufmann, 1997).

[12] D. Novillo, *Analysis and Optimization of Explicitly Parallel Programs*, Ph.D. thesis, Department of Computing Science, University of Alberta, Edmonton, Canada, 2000.

[13] V. Sarkar, Analysis and optimization of explicitly parallel programs using the parallel program dependence graph representation, *Lecture Notes in Computer Science*, (1366), Springer, 1997, 94-113.

[14] H. Srinivasan, J. Hook, & M. Wolfe, Static single assignment for explicitly parallel programs, *Proc. of the 20th ACM Symposium on Principles of Programming Languages (POPL)*, January 1993, 260-272.

[15] D. Shasha & M. Snir, Efficient and correct execution of parallel programs that share memory, *ACM Trans. Prog. Lang. Syst., 10*(2), 1988, 282-312.

[16] M. Weiser, Program slicing, *IEEE Transactions on Software Engineering*, (10), 1984, 352-357.