

1-1-1992

# Common Subexpression Induction?

H. G. Dietz

*Purdue University School of Electrical Engineering*

Follow this and additional works at: <http://docs.lib.purdue.edu/ecetr>

---

Dietz, H. G., "Common Subexpression Induction?" (1992). *ECE Technical Reports*. Paper 281.  
<http://docs.lib.purdue.edu/ecetr/281>

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact [epubs@purdue.edu](mailto:epubs@purdue.edu) for additional information.



# Common Subexpression Induction?

H. G. Dietz

TR-EE 92-5  
January 1992

---

† This work was supported in part by the Office of Naval Research (ONR) under grant number N00014-91-J-4013 and by the National Science Foundation (NSF) under award number 9015696-CDA.

## Table of Contents

<b>1. Introduction</b> .....	2
<b>2. Machine Characteristics for CSI</b> .....	2
<b>3. The CSI Algorithm</b> .....	4
<b>3.1. Example Code Segment</b> .....	4
<b>3.2. Algorithm Walk-Through</b> .....	6
3.2.1. Step 1: Construct Guarded DAG .....	6
3.2.2. Step 2: Inter-thread CSE .....	8
3.2.3. Step 3: Earliest and Latest Computation .....	9
3.2.4. Step 4: Classification .....	10
3.2.5. Step 5: Theoretical Lower Bound .....	12
3.2.6. Step 6: Creation of An Initial Schedule .....	13
3.2.7. Step 7: Improving the Initial Linear Schedule .....	13
3.2.8. Step 8: The Search .....	14
<b>3.3. Final Output for Simple Example</b> .....	15
<b>3.4. Partial Liveness</b> .....	16
<b>4. A Bigger; Tougher. Example</b> .....	17
<b>4.1. Recursive CSI</b> .....	19
<b>4.2. Simulated Annealing</b> .....	20
<b>5. CSI To Increase Vector Length</b> .....	20
<b>6. Summary and Conclusions</b> .....	22

# Common Subexpression Induction<sup>†</sup>

*H.G. Dietz*

Parallel Processing Laboratory  
School of Electrical Engineering  
Purdue University  
West Lafayette, IN 47906  
`hankd@ecn.purdue.edu`

## Abstract

SIMD (Single Instruction stream, Multiple Data stream) computers can only execute the exact same instruction across all processing elements. This paper presents a new **compiler** optimization that transforms multiple distinct code threads so that they have as many instructions in common **as** possible, hence, SIMD execution time is minimized. For example, SIMD "parallel **i f**" statements typically take the **then** clause time plus the **else** clause time to execute, but this new **transformation** usually can induce identical code sequences for most of the code in the **then** and **else** clauses, often yielding a 40% improvement in execution speed. The same principle also could be used to transform code which operates on multiple short vectors into operations on long vectors containing the catenation of the shorter vectors; for example, **operations** on two 8,192-element arrays might be combined into a single operation apparently acting on a 16,384-element array.

Keywords:: Common subexpression induction (CSI), common subexpression elimination (CSE), Single **Instruction** stream Multiple Data stream (SIMD), compiler optimization.

<sup>†</sup> This work was supported in part by the Office of Naval Research (ONR) under **grant** number N00014-91-J-4013 and by the National Science Foundation (NSF) under **award** number 9015696-CDA.

## 1. Introduction

**Traditional** compiler analysis and code transformation **are** based on **tracking** what happens to values, as noted in [CoS70]. For example, in common subexpression **elimination** (CSE) the compiler recognizes when the same value would be computed more than once and rewrites the code to **make** multiple references to a single computation of that value.

Dependence analysis [Ban88] and alias analysis [Die87] **are** different from the above in that they are **not** so much concerned with values as with storage locations. The compiler tries to recognize **when** references might access the same storage location, and can **parallelize** references in which different storage locations are accessed. Register, cache, and page **allocation/management** are also based on **tracking** storage locations.

The interesting point is that there is yet another type of program entity **which** can be analyzed: **the** code itself. A few compiler transformations, such as code straightening [CoS70], operate **directly** on the code structure, but relatively little attention has been given to this type of analysis **and** transformation.

**Common** subexpression induction (CSI)<sup>1</sup>, the topic of this paper, is the code-based equivalent of the value-based CSE optimization; CSI recognizes when the same code can be used for multiple execution threads. Whereas CSE is clearly beneficial when the execution time is directly **proportional** to the number of instructions executed, CSI is most beneficial **when** the execution time is **proportional** to the sum of the execution times for all control-flow **paths** — as in single **instruction** stream multiple data stream (**SIMD**) parallel computers.

The **following** section describes **the** machine properties that can make the CSI **optimization** useful. In section 3, each step of the CSI algorithm, and our prototype CSI tool, is explained. A simple example is used to illustrate the analysis and to expose an issue for **further** research involving the concept of register liveness for parallel machines. Section 4 presents a second, larger and less symmetrical, example. This second example is used to drive a discussion on how the algorithm from section 3 can be made still more efficient for large CSI problems. In section 5, we briefly present an example of how CSI can be used to increase apparent vector length, although we do not present an algorithm for this transformation. In closing, section 6 summarizes **the contributions** of this paper and directions for further study.

## 2. Machine Characteristics for CSI

The **basic** premise of CSI is that some machines have structures that permit a **single** instruction to **compute** several different values, hence, for those machines it is useful **to** be able to induce **code** structures that can maximize the number of useful values computed by each instruction. In **some** architectures, an improvement in parallelism results; in others, the **primary** effect is an improvement in cache performance. Machine architectures that can benefit from CSI include:

---

<sup>1</sup> **This name** was coined in [Die87], without a **practical** algorithm. It refers to the concept that, since some machines can have common subexpressions share the same code, it can be useful to induce such code sequences even if some additional instructions (e.g., register moves) must be inserted.

## **SIMD**

In a SIMD machine with  $N$  **PEs** (Processing Elements), up to  $N$  values **can** be computed by a single instruction. However, this performance can only be **achieved** if all  $N$  **PEs** (processing elements) will be executing **the** same operation — **PEs** that need to execute different operations cannot do useful work in that cycle. Because CSI increases the fraction of **PEs** that simultaneously execute the same instruction (the useful parallelism width), large speedups can be obtained. Examples of this type of machine include the TMC CM-2 [Thi90] and the **MasPar** MP-1 [Bla90].

## **Vector**

Although typically not as parallel, vector machines profit from CSI in **essentially** the same way that SIMD machines do. A good example of such a machine is the Cray **Y-MP** C90 [Cra91].

## **MIMD with shared I-Cache**

CSI, as described in this paper, is directly usable to improve **performance** of shared-memory MIMD (Multiple Instruction stream, Multiple Data stream) **systems** that have instruction caches with a mechanism for sharing. For example, **such** a sharing mechanism was proposed for the FMP [Lun87] and is generalized in [McC92].

The logic is simply that if a MIMD is programmed using the SPMD model (Single Program, Multiple Data), separate MIMD processes execute independently, but often are executing the same region of code at about the same time. Hence,, sharing an instruction cache can allow trailing processors to reuse the instructions  **fetched** by the leading processors. Clearly, by reducing the number of different control **flow** threads in the SPMD program, CSI can maximize the regions of code over which this sharing can occur.

## **VLIW and Superscalar**

Although VLIW (Very Long Instruction Word) [Fis84] and Superscalar **machines** can execute multiple operations within a single instruction, they also can benefit from a minor variation on the CSI optimization. The reason is simply that most IV-PE VLIW machines cannot pack  $N$  **arbitrary** operations into a single instruction — there are usually constraints on which operations can be packed together. For example, it is common to see a limit placed on how many **load/store** operations can **in** placed in each instruction. This optimization differs from CSI as described in this paper primarily in that the classification algorithm (see section 3.2.4) is **somewhat** more complex.

## **Sequential Nulling versus Jumps**

Some processors have instructions that allow the operation to be nulled **depending** on a condition code. For example, this is the mechanism used to implement "Squashing Branches" [McH86]. In such a serial machine, CSI can improve **performance** because it can replace branching overhead (both the **branch** instruction and the cache behavior it often introduces) with just a few instructions being marked as

nullable.

In this paper, we will focus on the application of CSI to massively parallel SIMD machines, in particular, to the **MasPar** MP-1. This is partly because the expected benefit to SIMD machines is very large, but also because the algorithm and examples are more easily understood. In addition, the **MasPar** MP-1 has hardware support for **PEs** to make indirect memory references, and this makes the **CSI** technique much more effective.

**Throughout** the rest of this paper, we use CSI to refer to CSI for a SIMD target **machine**.

### 3. The CSI Algorithm

The **CSI** algorithm analyzes a segment of code containing operations **executed** by any of multiple threads (enabled sets of SIMD **PEs**). From this analysis, it determines where threads can share the **same** code and what cost is associated with inducing that sharing. Finally, it generates a code schedule that uses this sharing, where appropriate, to achieve the minimum execution time. Unfortunately, this implies that the CSI algorithm is not simple.

Our prototype CSI tool implementation is also quite complex. It implements **only** CSI on assembly-level tuples — it is not a compiler and does not even perform final register allocation. Written in C using **PCCTS** [PaD92], the prototype consists of over 8,000 lines of C **source** code.

#### 3.1. Example Code Segment

In **order** to make the CSI algorithm more clear, the description of each major step in the algorithm is accompanied by a simple example code segment processed up to that stage in the CSI algorithm. The example code is not particularly meaningful, but clearly demonstrates the algorithm. The code is:

```

if (parallel-expression) {
    /* Then clause */
    c = a + b;
} else {
    /* Else clause */
    c = a - b;
}

```

this high-level C-like parallel code corresponds to assembly-level code like:

**"then" clause**

```
c = a + b;
```

0	const	#a
1	load	0
2	const	#b
3	load	2
4	add	1,3
5	const	#c
6	store	5,4

**else clause**

```
c = a - b;
```

0	const	#a
1	load	0
2	const	#b
3	load	2
4	neg	3
5	add	1,4
6	const	#c
7	store	6,5

In executing this code, first the value of parallel-expression would be computed on all currently enabled **processing** elements (**PEs**). Next, the set of enabled processors would be masked down to only **those** for which parallel-expression evaluates as true. Only these **PEs** would execute `c = a + b;`. Having completed the "then" clause, the **SIMD** machine would **prepare** to execute the **else** clause by changing the enable mask so that only **PEs** whose *parallel\_expression* is false are enabled. After these **PEs** have executed `c = a - b;`, the enable mask is restored to its state prior to entering the **if**. Hence, the time taken within the **if** statement clauses is essentially the time for the "then" clause + the time for changing the enable mask + the time for the **else** clause.

In **contrast**, the CSI optimization attempts to bring the execution time as close as possible to *maximum*("then" time, **else** time), which would be the time taken if both clauses could be executed simultaneously without masking overhead.

Since the CSI optimization is explicitly based on minimizing execution time, we also need to associate a cost with each operation. In this paper, we use the approximate execution times of the **instructions** counted in units of machine cycles for a MasPar MP-1 [Bla90]. Note, however, that our **instructions** do not match those of the **MasPar** and we do not model the overlap that the **MasPar** allows between memory references and other PE operations. Hence, these **times** are realistic, but **only** approximately correspond to MasPar times.

Given that disclaimer, the execution time for the above code is:

Then time		615
Mask time	+	9
Else time	+	639
		<hr/>
		1263 clock ticks

and our goal is to use CSI to reduce the time to be as close as possible to *maximum*(615, 639), or 639 clock ticks. If this goal is achieved, the execution time of the code would be reduced by a

factor of 49%.

## 3.2. Algorithm Walk-Through

Sections 3.2.1-3.2.8 detail each major step in the CSI algorithm as it is **currently** implemented in our prototype CSI tool. The state of the above example is given with each step's description.

The **algorithm** can be summarized as follows. First, a guarded DAG is constructed for the input, then this DAG is improved using inter-thread CSE. The improved DAG is **then** used to compute **information** for pruning the search: earliest and latest, operation classes, anti theoretical lower **bound** on execution time. Next, this **information** is used to create a linear **schedule** (SIMD execution sequence), which is improved using a cheap approximate search and then used as the initial schedule for the **permutation-in-range** search that is the core of the CSI optimization.

### 3.2.1. Step 1: Construct Guarded DAG

The first step in the CSI algorithm is the construction of a guarded DAG for the **assembly-level** operations. The use of a DAG, Directed Acyclic Graph, to represent data dependencies has long been a standard technology for optimizing compilers [AhS86]; however, the **concept** of a guarded **DAG** is somewhat unusual. Normally, each node in a DAG represents an **operation** and each arc **represents** data flowing between operations. It is assumed that *every operation is executed if any operation is executed*. This is not true for the code of a traditional **if** statement.

However, in the SIMD view, every operation within a parallel **if** is executed if any operation is executed<sup>2</sup>. The catch is that operations may be executed by different sets of **PEs**. Hence, we need some way of tracking which **PEs** will execute which instructions.

We do this by associating a unique guard value with alternative path selected **by** each conditional expression in the code segment. These guard values are then encoded as individual bits. It is then possible to tag each **instruction** with a guard which is the "or" of the **guard** values that dominate its execution. In this way, code with arbitrary forward branching can be analyzed as a single **DAG** in which each node is tagged with its guard value. For example, consider the guard markings **in** listing 1.

---

<sup>2</sup> Some SIMD languages implicitly perform a test to see if any PE will be enabled and jump over the clause if none will be enabled. For example, this is the definition used by MPL for parallel **if** statements [Mas91]. Here, we assume that no such test and jump is made.

```

/* guard 1|2|4|8|16 */
if (parallel-expression1) {
  /* guard 1|2|4|8 */
  switch (parallel-expression2) {
    case A: /* guard 1 */ break;
    case B: /* guard 2 */
    case C: /* guard 2|4 */ break;
    default: /* guard 8 */ break;
  }
  /* guard 1|2|4|8 */
} else {
  /* guard 16 */
}
/* guard 1|2|4|8|16 */

```

Listing 1: Example of Guard Labeling

**Neither** multi-way branches (`parallel cases`) nor nested conditionals is a problem. Loops, which are **formed** by backward branches, require that the inside of the loop be handled as a separate problem from the code before and after the loop. A similar difficulty **occurs** when independent conditional statements are executed in a sequence, rather than nested; in such a case, the easiest solution is to analyze the conditionals separately<sup>3</sup>. Although the **examples** in this paper have no more than two threads (guard bits), the current prototype **CSI** tool can **process** arbitrary **guarded** input with up to 32 threads.

**Returning** to our example `if` statement, the result is the guarded DAG of **figure 1**

---

<sup>3</sup> It is actually better to modify the guard handling so that this case can be analyzed intact, but that is much more complex, and has been omitted from the algorithm in this paper.

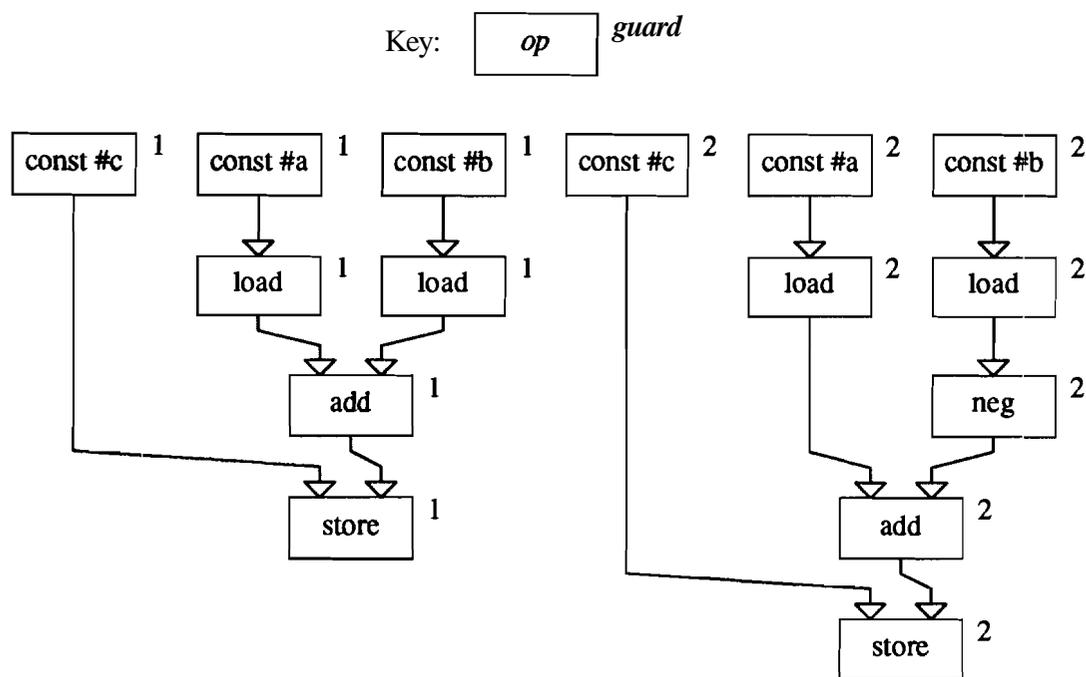


Figure 1: Original Guarded DAG for Example

### 33.2. Step 2: Inter-thread CSE

Given a guarded DAG, the next step is very similar to recognizing and **factoring-out** "common subexpressions." However, it is not quite traditional CSE, because **operations with different guards can be factored-out as common subexpressions**. Hence, we call this step "inter-thread CSE," although the effect is more like a combination of **conventional CSE** and code hoisting (except we don't need a dominator or code motions).

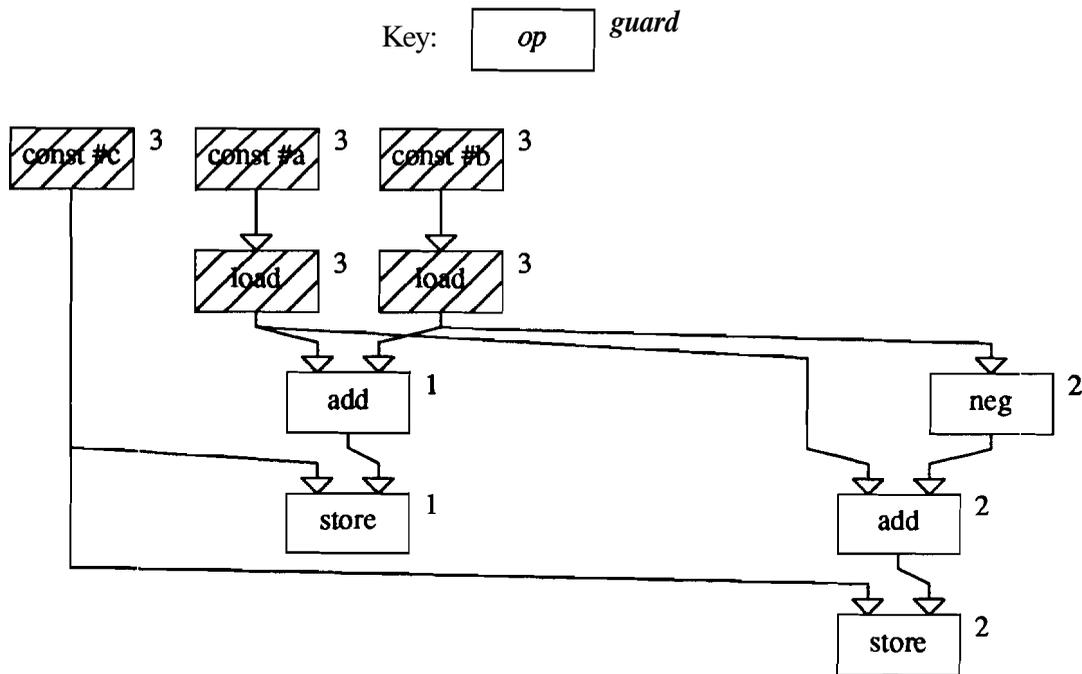
Whereas traditional CSE recognizes when two computations would produce the same value, inter-thread CSE recognizes when two computations would produce the same value **if** they were executed **by** the same processor. For each inter-thread common subexpression, the remaining operation is given the guard that is the "or" of the guards for all operations **absorbed** by that optimization.

This works because, even though operations with different guards may be executed on different **PEs**, the instruction sequence that combines one **PE's** local data in a **particular** way must **perform** the same function for another PE working on its local data. If the guards have bits in common, it simply means that traditional CSE was performed on some **PEs**.

After inter-thread CSE, cost of the example drops from 1263 to 891 clocks<sup>4</sup>; a 29%

<sup>4</sup> Actually, 891 is the cost obtained after conversion to a linear schedule in step 6 (section 3.2.6), but this number reflects only the benefit gained by the use of inter-thread CSE.

reduction. The resulting guarded DAG is given in figure 2.



**Figure 2:** Guarded DAG After Inter-Thread CSE

### 3.2.3. Step 3: Earliest and Latest Computation

After performing inter-thread CSE, there are a few search pruning characteristics that need to be computed before the CSI search phase is begun. Perhaps the most basic of these are earliest and latest.

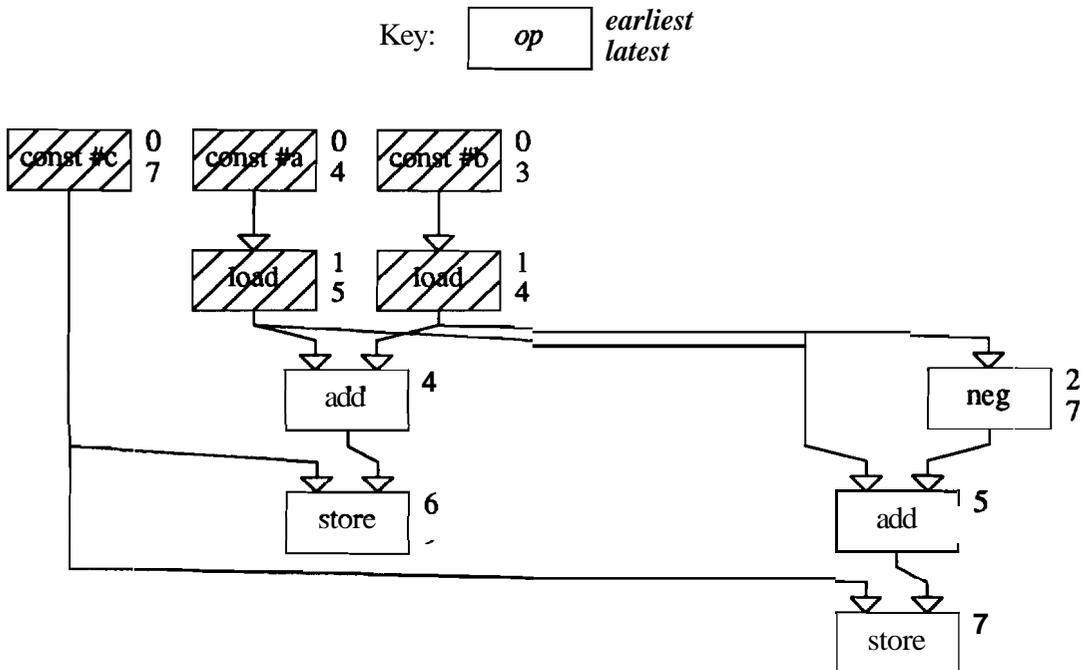
The CSI search operates on a linear schedule of the instructions, rearranging that linear schedule and considering combining only those instructions which are adjacent in the linear schedule. This implies a permutation search. The problem is simply that even a small CSI example, such as the one used in this paper, would yield too large a search space if a complete permutation search was used. Using a full permutation search on the small example in this paper would require consideration of  $10!$ , or 3,628,800, linear schedules; the larger example given in section 4 contains 23 instructions, hence,  $23!$  (25,852,016,738,884,976,640,000) different schedules would need to be examined. Without very effective pruning, CSI is infeasible.

One of the most effective pruning methods is to simply eliminate the linear schedules that violate the precedences expressed by the DAG — for example, any linear schedule that places the **neg** operation before **const #b** is invalid and need not be considered. The problem is that to check each schedule for validity using the DAG is relatively expensive because we would still have to generate the bad schedules in order to check them. The earliest and latest measures provide a way of performing a somewhat conservative version of the DAG check without actually

generating the schedules that would fail the test.

Earliest for an operation is the earliest position in the linear schedule which that operation could **occupy** without violating the DAG, in other words, it is the number of DAG predecessors. Latest is **the** latest viable position for that operation in the linear schedule, which is **equivalent** to the total number of operations minus the number of operations which have that operation as a predecessor (including the operation itself). Rather than performing an ordinary permutation search on **the** linear schedule, a permutation-in-range can be used, restricting each operation to move only through slots in its earliest to latest range.

For our example, the result of earliest and latest labeling is shown in figure 3.



**Figure 3:** Earliest and Latest Labeling of the DAG

We **applied** a similar technique to reduce the search space for generation of optimal code schedules for pipelined machines. An **overview** of this code scheduling technique applied to pipelined **machines** appears in [NiD90]; a more detailed treatment, including a proof that optimal schedules are obtained, is given in [Nis90].

#### 32.4. Step 4: Classification

Just as the earliest and latest information is used to prune a search, it can save a significant amount of time if operations are grouped into classes prior to the start of the search.. Each class consists of a set of operations such that it is allowable for each operation in **that** class to be merged with at least one other member of that class. Hence, if two operations are not in the same

class, then: is no need for a more detailed (and more expensive) check to determine if they could be merged.

**Classes** are formed to be as small as possible so that for each class:

1. The **opcodes** for all members of this class are the same.
2. The immediate operands, if any, for all members of this class are the same.
3. The class members cannot be partitioned such that the operations in some partition element all must execute after all the operations in some other partition element. Using the DAG, this is quite complex to check; hence, we use a **conservative** approximation.

If the members of a class are sorted by earliest as the primary key and latest as the secondary key, one can simply check that each pair of adjacent operations in the class have overlapping **earliest..latest** ranges. If the ranges do not overlap, then the class can be partitioned into two classes by splitting it between the nonoverlapping adjacent operations.

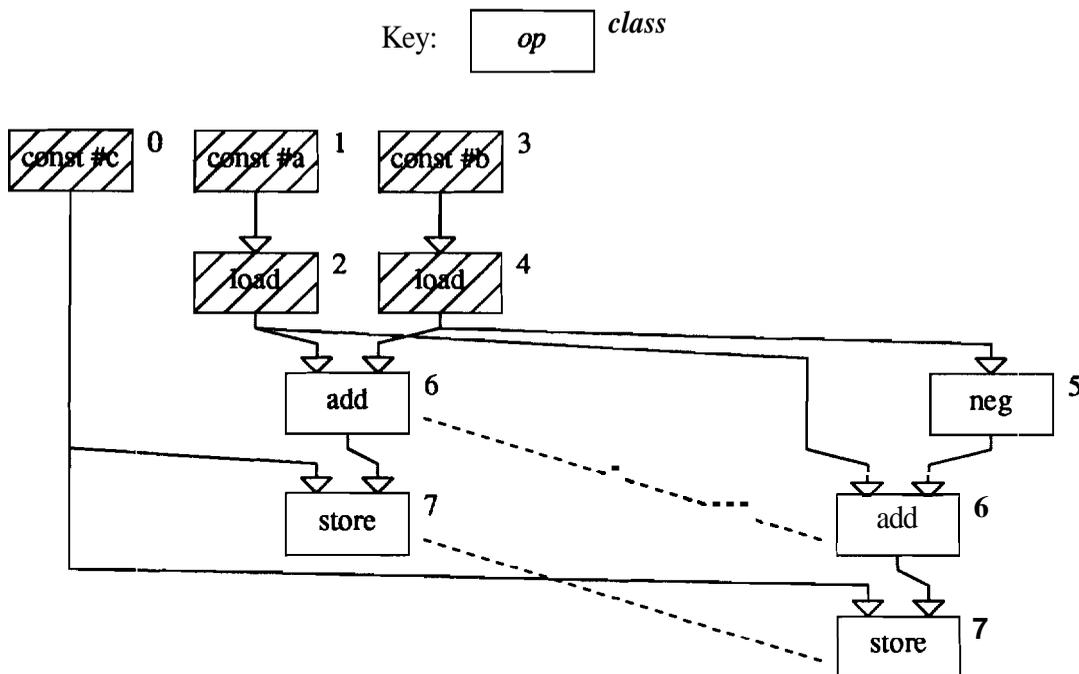
4. Every operation whose guard covers all other guards within its class can **be** made into a singleton class.

As a simple approximation to this, we used the rule that an operation whose guard is all threads is a singleton class.

5. All members of this class do not have a thread in common. If they do, **each** should **be** its own singleton class.

The first **two** conditions are a direct consequence of basic SIMD execution: the same information must be **broadcast** to all **PEs**. Condition 3 reduces classes by applying DAG **constraints**. The 4th and 5th **conditions** actually follow from the observation that after CSE, no two **instructions** that can be executed by the same thread can be merged; if they could be, they **would** have been factored-out when inter-thread CSE was performed.

The **class** formation procedure simply applies rules 1 and 2 to create initial class groupings and then recursively attempts to reduce these classes using rules **3, 4,** and 5. The **result** for our example **code** is show in figure 4.



**Figure 4:** Classification of Operations in the **DAG**

### 3.2.5. Step 5: Theoretical Lower Bound

Using the classes and expected execution times for each type of operation, it is possible to compute a good estimate of the lower bound on minimum execution time. This estimate can be used to **determine** if performing the CSI search is worthwhile — i.e., if the potential for improvement in code execution time by CSI is small, then one might abort the search. The same algorithm is **used** to evaluate partial schedules to aid in pruning the search.

The estimate is computed by:

1. For each class, group members together if the guards do not overlap.
2. The cost for each class is the number of members remaining times the cost of that operation.

This bound might be unachievable because it ignores detailed dependence constraints (**DAG** checks) **and** it ignores the cost of masking, but it is quick to compute and the estimates are usually very close to the best achievable execution time.

The **minimum** number of instructions remaining after **CSE**, or minimum "ticks", is also computed at this stage.

For the simple example `if` statement, the computed lower bound is **639** clock ticks, resulting in a total of 8 instructions after **CSI**.

### 3.2.6. Step 6: Creation of An Initial Schedule

Before the search can be performed, the code must be converted into a linear schedule. In the linear **schedule**, the Nth operation in a schedule is either executed at the same **time as** the (N-1)th instruction or in the next "tick." There are two reasons that a linear schedule is used:

1. The permutation-in-range search (step 8, section 3.2.8) is relatively efficient using the linear schedule.
2. The linear schedule corresponds to a SIMD execution sequence, and **this** instruction sequence must be examined in order to compute an accurate cost. There is a great temptation to view CSI as a graph node matching problem on the DAG, but combining some graph nodes implies a significant cost which is not computable without the linear order.

When two nodes (operations) are combined, it might not be possible for **the** operands to be directly placed in the same registers under both original guards. Hence, it may be necessary to insert one or more register-to-register moves that would be executed under one of the guards. This may, in turn, involve additional cost for masking — unless the register-to-register move can be executed immediately before or after another instruction that has the same guard. Since these costs depend on **properties** of the SIMD (linear) schedule, and these costs can easily outweigh the benefit of combining, the linear schedule must be examined.

Hence, in this step we convert the DAG into a linear schedule.

The linear schedule is created by performing a level-order traversal of the **DAG**, but in this traversal **operations** in the same class **as** the previous operation in the schedule are given preference. This tends to group together instructions that could be combined by CSI.

### 3.2.7. Step 7: Improving the Initial Linear Schedule

Although any linear schedule that does not violate the DAG constraints would be valid **as** input to the search, cost pruning is used and finding a better schedule earlier will cause more pruning. **This** makes it worthwhile to invest a little effort in making the initial **schedule** relatively **good**.

**Currently**, the prototype CSI tool performs a "sort" of the schedule generated in step 6. Instead of using comparison of key values, in this "sort," elements in the schedule exchange places **only** if the exchange reduces a very crude estimated cost. This portion of the **CSI** tool is a "hack," but it does tend to significantly improve the schedule, and hence improves pruning in the "real" search.

At this stage, the code for the simple `if` example has been restructured so **that** it requires only 690 ticks. This represents an additional reduction of 23% over the improvement due to inter-threat1 CSE, or a total improvement of 45% as compared to the original code.

### 3.2.8. Step 8: The Search

Given the information determined in the previous steps, we are now ready to perform the **permutation-in-range** search for the minimum execution time schedule. The technique presented here is very similar to that which we used in code scheduling for pipelined machines [NiD90][Nis90], except in that the pruning and cost criteria **are** different and we have the extra dimension of considering merges of adjacent instructions.

For the search, the initial ***N*-operation** linear schedule is partitioned into two parts: the *n*-operation partial schedule under evaluation (schedule slots **0..*n*-1**) and the portion of the schedule that has not yet been evaluated (***n*..*N*-1**). The basic step in the permutation-in-range search is to consider swapping the instruction in slot *n* with any of the instructions in slots **greater** than *n*. Whenever a viable swap is found, the incremental change to the partial schedule is evaluated. A viable swap causes the partial schedule to be extended by moving the partition to between slots *n* and ***n*+1**; a swap that cannot lead to a better complete schedule prunes all schedules with that ***n***-operation prefix.

The **main** components of the search are:

1. Only consider swaps for which the instruction being swapped into the partial schedule at position ***n*-1** has **earliest**  $\leq n-1$  and the instruction being swapped out has **latest**  $\geq n$ . Note that if this condition is not met, then not only is the swap disallowed, but additional pruning is possible.
2. Only swaps that do not violate the DAG precedences are valid.
3. As each operation is added to the partial schedule, it might either execute in the tick after the previous operation. Alternatively, if it can merge with that operation, it would execute in the same tick. Merges **are permitted** only if the operations are in the same class and there **are** no DAG or guard conflicts (**i.e.**, no instruction being merged is the predecessor of any other instruction being merged and none of the **guards** overlap).
4. In a machine (like the **MasPar**) in which combining usually is **beneficial**, give precedence to swapping-in operations that are of the same class as the previous operation.
5. For much the same reason given in rule 4, when a merge of instructions **into** the same tick is possible, the merger is evaluated before the non-merged schedule.
6. Because merging happens with adjacent operations in the linear schedule, if there are *k* instructions that can merge into one, there **are** *k*! different possible orderings in which they might appear with the same result. This would multiply the search time by *k*!. Hence, merges **are** only allowed if the operations being merged **are** in order of increasing internal identifier. For example, merging tuples 4 and 12 (the **add** operations in the example) will be allowed only if their order in the linear **schedule** is (4, 12), not if their order is (12, 4).

This reduces the search space equivalently to using ticks, rather than **schedule** slots, for the linear schedule. However, since the number of slots is fixed and **the** number of

ticks varies, the slot scheme with this adjustment yields a more efficient **search**.

7. A swap that must result in a schedule worse than or equal to the best found thus far need not be investigated further. Hence, if the cost of the current partial schedule + the theoretical minimum cost of the operations remaining to be scheduled  $\geq$  cost of the best complete schedule found thus far, the swap is considered invalid.

Note that computing the cost involves more than just observing whether a merge is possible; it is also necessary to compute the approximate overhead in placing operands in the same registers for the merged operations. The masking and register move cost computation used in this paper is simply that each operand that cannot be trivially renamed to be in the appropriate register adds the cost of one register-to-register move + one mask operation unless the previous instruction **has** the same guard. This is a gross oversimplification of how it should really work (see section 3.4), but the ideal register allocation process is too complex to describe in this paper and the method described here produces acceptable results.

**All** of these techniques have the property that they will never prune a unique optimal schedule. Hence, if **allowed** to run to completion, the technique is equivalent to an **exhaustive** search and ensures that the optimal schedule will be found.

Despite the pruning, **running** to completion is not always feasible. We suggest that in such cases an **upper** limit should be placed on the number of operation swaps considered. That limit could be a fixed number or, perhaps more useful in practice, it could be derived based on the level of optimization specified by the programmer and the amount of potential **improvement** estimated by the theoretical bound.

### 33. Final Output for Simple Example

After the search has completed (or been artificially terminated before completion), the resulting linear schedule is the **SIMD** program. In the version of the **CSI prototype** described here, the **SIMD** program need only have registers assigned and masking and register move code inserted.

The linear schedule output by the **CSI** prototype tool for the simple **if** example is:

```

;Initial cost = 1263
;Cost after inter-thread CSE = 891
;Theoretical lower bound ticks = 8
;Theoretical lower bound cost = 639
;At perm #11, new cheapest is 690...
;At perm #21, new cheapest is 666...
;Final Tuples (651 perm calls, cost 666):
code
3:0    const  #a    ;tick 0
3:2    const  #b    ;tick 1
3:5    const  #c    ;tick 2
3:1    load   0     ;tick 3
3:3    load   2     ;tick 4
2:11   neg    3     ;tick 5
1:4    add    1,3   ;tick 6
2:12   add    1,11  ;tick 6
1:6    store  5,4   ;tick 7
2:14   store  5,12  ;tick 7

```

The **format** is *guard: operation :tick*. Notice that the search ran to completion in just 651 swaps (1 **swap**  $\equiv$  1 perm call) and only the **Ne g** instruction is not executed by all **PEs**. The result is 47% faster than the original **code**.

**However**, there is also an unpleasant little surprise: the ideal execution time was not achieved. **The** execution time is 666 clocks when it should have been 639. The reason has to do with a **nasty** little problem concerning register allocation and the concept of "register liveness."

### 3.4. Partial Liveness

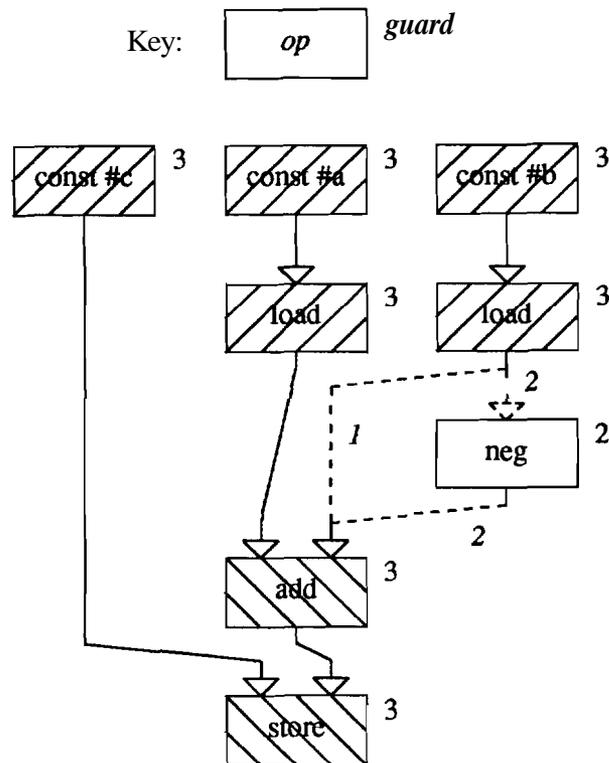
In a conventional machine, a register either holds a live value or it is free for reuse. In a **SIMD machine** (or any parallel machine), a register can be live or dead for any guard, and can be **simultaneously** live with different values in different threads. We call this new concept "partial liveness" **and** it is responsible for the difference between 666 and 639 clocks for our simple example.

To **better** understand this, consider the DAG showing the final state of the **example** (see figure 5). Notice that the register holding the result of loading **b** is used in two **places**: by **Ne g** and by **Add**. In the linear schedule (see above), the **Ne g** instruction comes *before* the **Add**. Hence, **when** allocating a register for the result of the **Ne g** instruction, **conventional** liveness analysis finds that the register holding the loaded value of **b** is still live and that register cannot be reused **for** the result of the **Ne g**. Therefor, the result of **Ne g** is placed in another register... until **registers** are assigned for the merged **Add** that uses the value.

In **order** to merge **Add** operations 4 and 12, a **register-to-register** move is **inserted** to move the result of **Ne g** into the same register that holds the loaded value of **b** on the other thread. 666 is simply 639 + the move overhead.

Had our CSI tool been smarter, it would have realized that the register holding the loaded value of `b` is only partially live after the `Neg` instruction, hence, it could have been reused without **conflict**. That knowledge would have allowed it to achieve the **639** clock theoretically optimal **time**; an execution time reduction of **49%**.

Unfortunately, partial liveness in the context of CSI becomes much more complex as larger codes are considered, hence, it will have to be the topic of a future paper. In this paper, we assume the traditional definition of liveness — and suffer the penalty.



**Figure 5: DAG Showing Final State of Example**

#### 4. A Bigger, Tougher, Example

**While** the example case used to illustrate the CSI algorithm obtained a good **speedup**, it is not clear **how** often the code sequences for different threads will look that similar. Neither is it clear that **any** performance is gained when the threads differ more significantly.

We **do** not have statistics available on how often threads have very similar code, although it seems **fairly** likely that SIMD code involving tests for "edge conditions" would have this property. To **answer** the question of how performance degrades with larger, less symmetrical, code, we present the following example.

The **following** gives sample code for `then` and `else` clauses that differ **significantly**. For example, only the lvalue of `a` and the rvalue of `c` are inter-thread CSEs. The `then` clause even has **one** more memory reference than the `else` clause.

<b><code>then</code> clause</b>			<b><code>else</code> clause</b>		
<code>a = *b + c;</code>			<code>a = c + e;</code>		
<code>d = a + d;</code>			<code>e = f - g;</code>		
0	const	#b	0	const	#c
1	load	0	1	load	0
2	load	1	2	const	#e
3	const	#c	3	load	2
4	load	3	4	add	1,3
5	add	2,4	5	const	#a
6	const	#a	6	store	5,4
7	store	6,5	7	const	#f
8	const	#d	8	load	7
9	load	8	9	const	#g
10	add	5,9	10	load	9
11	store	8,10	11	neg	10
			12	add	8,11
			13	store	2,12

The **interesting** result is that CSI works nearly as well as it did on the simple **example**. This is **primarily** because the **MasPar** supports indirect memory references, so **all** memory references can be merged. Such merges are usually profitable because the PE local memory interfaces on the **MasPar MP-1** are shared by groups of PEs [Bla90], often making memory **reference** time the performance-limiting factor. In addition, performance is helped by the fact that **enable** masking and register-to-register moves are both quick operations.

Listing 2 gives the complete output from the CSI prototype tool. The initial code would have taken 2478 clocks. Inter-thread CSE by itself would only have reduced that by 7%, to 2312; **after** the **sort** described in step 7 (section 3.2.7), the reduction would have been just 13%, to 2159. However, **the** full CSI algorithm gives an impressive performance, reducing the time to just 1386 clocks — a 44% reduction.

Unlike the simple example, in this case the search did not **run** to completion, so optimality is not guaranteed. The algorithm examines swaps at a rate of about 20μs/swap **running** on a SPARC server, and was allowed to **run** for 1,000,000 swaps (20 seconds). A total of just 21 complete **schedules** were considered — this should be contrasted with the 23! possible schedules.

```

;Initial cost = 2478
;Cost after inter-thread CSE = 2312
;Theoretical lower bound ticks = 16
;Theoretical lower bound cost = 1259
;At perm 124, new cheapest is 2159.. .
;At perm #44, new cheapest is 1990.. .
;At perm 151, new cheapest is 1981.. .
;At perm #99, new cheapest is 1976.. .
;At perm #106, new cheapest is 1803.. .
;At perm #117, new cheapest is 1789.. .
;At perm #1921, new cheapest is 1788...
;At perm 11932, new cheapest is 1774.. .
;At perm 13646, new cheapest is 1765.. .
;At perm #38353, new cheapest is 1756...
;At perm #61413, new cheapest is 1747.. .
;At perm 1120831, new cheapest is 1602...
;At perm 1120842, new cheapest is 1588...
;At perm 1122646, new cheapest is 1587...
;At perm 1122657, new cheapest is 1573.. .
;At perm #124371, new cheapest is 1564.. .
;At perm #188984, new cheapest is 1555...
;At perm #242020, new cheapest is 1415.. .
;At perm #242031, new cheapest is 1401...
;At perm #243835, new cheapest is 1400.. .
;At perm #243846, new cheapest is 1386.. .
;Final Tuples (100000 perm calls, cost 1386):

```

	code			
	1:0	const	#b	;tick 0
	3:3	const	#c	;tick 1
	3:6	const	#a	;tick 2
	1:8	const	#d	;tick 3
	2:14	const	#e	;tick 4
	2:19	const	#f	;tick 5
	2:21	const	#g	;tick 6
	1:1	load	0	;tick 7
	2:22	load	21	;tick 7
	1:9	load	8	;tick 8
	2:20	load	19	;tick 8
	1:2	load	1	;tick 9
	2:15	load	14	;tick 9
	3:4	load	3	;tick 10
	2:23	neg	22	;tick 11
	1:5	add	2,4	;tick 12
	2:16	add	15,4	;tick 12
	1:10	add	5,9	;tick 13
	2:24	add	20,23	;tick 13
	1:7	store	6,5	;tick 14
	2:18	store	6,16	;tick 14
	1:11	store	8,10	;tick 15
	2:25	store	14,24	;tick 15

Listing 2: Final Output for Tougher Example

Although it is mildly disappointing that the search could not run to **completion** for this or other large test cases, we have found performance to be consistently good even when the search is truncated. The prototype **CSI** tool was even effective in helping to optimize **SIMD** programs that had over a hundred instructions and many threads [DiC92].

#### 4.1. Recursive CSI

While good performance was obtained using the **CSI** algorithm in this paper, still better pruning **would** be desirable. One obvious approach is to partition the original **CSI** problem into two or **more** parts, schedule each independently from the others, and then apply the **CSI** algorithm to the concatenation of the schedules for each part.

The **CSI** prototype implementation does not automatically provide this **recursive** subdivision, but **can** read its output as input. Hence, we were able to perform a simple **experiment** by hand-partitioning the original code into two parts, using **CSI** on each, and then using **CSI** on the concatenation of the two outputs. Although essentially the same final schedule was **obtained**, the recursive application did cause a faster pruning, and the number of swaps totaled for all three **CSI** runs was **less** than that for the single **CSI** run over the complete initial code.

The problem is that the improvement in search speed by recursive subdivision is critically dependent on choice of partitioning, and we do not yet have a good method by which the partitions can be mechanically generated.

#### 4.2. Simulated Annealing

Another possible way to speed convergence of the search is to modify the driver from section 3.2.8 to use a simulated annealing approach. Notice that **all** the pruning methods can still be applied, but the benefit would be somewhat less than in the current search. *All* pruning in the simulated annealing would be pruning complete schedules, whereas permutation-in-range can incrementally prune a partial schedule and all complete schedules that contain it.

Unfortunately, we have not yet been able to implement a simulated annealing in our prototype and determine its effectiveness.

#### 5. CSI To Increase Vector Length

Thus far, this paper has discussed CSI as a method to improve the execution **speed** of SIMD conditionals. In this section, we suggest that the same technology, combined with careful data layout, is also the key to creating long vector operations out of short vector operations, or even vector **operations** out of scalar references.

**Suppose** one has a 16,384-PE machine and SIMD code:

```

i n t  a1[8192], a2[8192];
i n t  b1[8192], b2[8192];
i n t  c1[8192], c2[8192];

c1 = a1 + b1;
c2 = a2 - b2;

```

To get the best memory utilization, this should result in a memory layout like that shown in figure 6.

PE0	PE1	PE2		PE8191	PE8192	...	PE16383
a1[0]	a1[1]	a1[2]		a1[8191]	a2[0]		a2[8191]
b1[0]	b1[1]	b1[2]		b1[8191]	b2[0]		b2[8191]
c1[0]	c1[1]	c1[2]		c1[8191]	c2[0]		c2[8191]

**Figure 6:** Memory Layout for 8,192-Element Arrays

However, given 16,384 **PEs**, it makes sense to imagine that each memory **object** is 16,384 elements in width. This renaming of the memory cells gives the layout depicted in figure 7.

PE0	PE1	PE2		PE8191	PE8192	...	PE16383
a[0]	a[1]	a[2]	...	a[8191]	a[8192]	...	a[16383]
b[0]	b[1]	b[2]	...	b[8191]	b[8192]	...	b[16383]
c[0]	c[1]	c[2]	...	c[8191]	c[8192]	...	c[16383]

**Figure 7:** Memory Layout for 8,192-Element Arrays

This is interesting because reflecting this renaming back into the source program yields:

```

int a[8192 + 8192];
int b[8192 + 8192];
int c[8192 + 8192];

if (PE_number < 8192) {
    c = a + b;
} else {
    c = a - b;
}

```

which is **exactly** the same `if` statement that was used for the simple example in section 3.1 of this paper.

**Admittedly**, there is much work to be done before CSI can be combined with **sophisticated** data layout to mechanically lengthen vectors, but this gives a clear direction for **future** research.

## 6. Summary and Conclusions

CSI was originally proposed in [Die87], but no practical algorithm had been found until April 1991. The algorithm is not simple, and can certainly be improved further, but our prototype **implementation** has shown CSI to be both feasible and surprisingly effective in at least a few test cases.

In **some** sense, CSI is the most fundamental compiler transformation for a **SIMD**, because it merges **threads** to **keep PEs** enabled. It does this by merging instructions from different paths within `then` and `else` clauses, multiway branches, and even nested **conditionals**. Coupled with new techniques for data layout, it should also be possible to use CSI to create "vectors" out of groups of ordinary scalars, and longer vectors out of multiple short vectors.

The **CSI** algorithm given in this paper clearly could benefit from further study and refinements, and some improvements are suggested in the paper. However, the **prototype** implementation has highlighted an important defect in current compiler technology for parallel machines: the inappropriateness of using ordinary liveness for register allocation. **As** a solution, we have **introduced** the concept of "partial liveness" to more accurately manage register usage, especially in SIMD machines.

Finally, it is useful to recall that variations on CSI apply to a fairly wide range of architectures (see section 2), not just SIMD. Perhaps the generality of CSI will lead to research on other new compiler transformations based on analysis of code (operations), rather than the far more common **analysis** of values (data flow analysis) or of storage locations (dependence **analysis**)?

## References

- [AhS86] A.V. **Aho**, R. **Sethi**, and J.D. **Ullman**, *Compilers: Principles, Technique., and Tools*, Addison-Wesley, Reading, Massachusetts, 1986.
- [Ban88] U. **Banerjee**, *Dependence Analysis for Supercomputing*, Kluwer, Boston, Massachusetts, 1988.
- [Bla90] T. Blank, "The MasPar MP-1 Architecture," 35th IEEE Computer Society International Conference (**COMPCON**), pp. 20-24, February 1990.
- [CoS70] J. **Cocke** and J.T. **Schwartz**, *Programming Languages and Their Compilers*, Second Revised Version of Course Notes, Courant Institute, New York University, New York.
- [Cra91] Cray Research Incorporated, *The CRAY Y-MP C90 Supercomputer System*, Eagan, Minnesota, 1991.
- [DiC92] H.G. Dietz and W.E. Cohen, "A Massively Parallel MIMD Implemented By SIMD Hardware," submitted to the 1992 International Conference on Parallel **Processing**.
- [Die87] H.G. Dietz, *The Refined-Language Approach to Compiling for Parallel Supercomputers*, Ph.D. Dissertation, Polytechnic Institute of New York, Brooklyn., New York, 1987.
- [Fis84] J.A. Fisher, "The VLIW Machine: A Multiprocessor for **Compiling** Scientific Code," IEEE Computer, pp. 45-53, July 1984.
- [Lun87] S. F. Lundstrom, "Applications Considerations in the System Design of Highly Concurrent Multiprocessors," *IEEE Trans. Comput.*, vol. C-36, no. 11, pp. 1292-1309, Nov. 1987.
- [Mas91] MasPar Computer Corporation, *MasPar Programming Language (ANSI C compatible MPL) Reference Manual, Software Version 2.2*, Document Number 9302-0001, Sunnyvale, California, November 1991.
- [McH86] S. **McFarling** and J. Hennessy, "Reducing the Cost of Branches," 13th Symposium on Computer Architecture, pp. 396-403, June 1986.
- [MeO92] J.C. **Mejia** and M.T. **O'Keefe**, "High Performance Instruction **Memory** Design for Multiprocessors," preprint of paper submitted to the 1992 International Conference on Parallel Processing.
- [NiD90] A. Nisar and H.G. Dietz, "Optimal Code Scheduling for Multiple **Pipeline** Processors," 1990 International Conference on Parallel Processing, vol. **II**, Saint Charles, Illinois, pp. 61-64, August 1990.
- [Nis90] A. Nisar, *Optimal Code Scheduling for Multiple Pipeline Processors*, Master's Thesis, Purdue University, West Lafayette, Indiana, 1990.
- [PaD92] T.J. **Parr**, H.G. Dietz, and W.E. Cohen, "**PCCTS** Reference Manual (version 1.00)," ACM SIGPLAN Notices, accepted to appear, February 1992.

[Thi90] **Thinking Machines Corporation**, *Connection Machine Model CM-2 Technical Summary, " version 6.0, Cambridge, Massachusetts, November 1990.*