Optimizing Memory Accesses For Spatial Computation

Mihai Budiu and Seth C. Goldstein
Computer Science Department
Carnegie Mellon University
{mihaib,seth}@cs.cmu.edu

Abstract

In this paper we present the internal representation and optimizations used by the CASH compiler for improving the memory parallelism of pointer-based programs. CASH uses an SSA-based representation for memory, which compactly summarizes both control-flow- and dependence information.

In CASH, memory optimization is a four-step process: (1) first an initial, relatively coarse, representation of memory dependences is built; (2) next, unnecessary memory dependences are removed using dependence tests; (3) third, redundant memory operations are removed (4) finally, parallelism is increased by pipelining memory accesses in loops. While the first three steps above are very general, the loop pipelining transformations are particularly applicable for spatial computation, which is the primary target of CASH.

The redundant memory removal optimizations presented are: load/store hoisting (subsuming partial redundancy elimination and common-subexpression elimination), load-after-store removal, store-before-store removal (dead store removal) and loop-invariant load motion.

One of our loop pipelining transformations is a new form of loop parallelization, called loop decoupling*. This transformation separates independent memory accesses within a loop body into several independent loops, which are allowed dynamically to slip with respect to each other. A new computational primitive, a* token generator *is used to dynamically control the amount of slip, allowing maximum freedom, while guaranteeing that no memory dependences are violated.*

1 Introduction

One of the main bottlenecks to increasing the performance of programs is that many compiler optimizations break down in the face of memory references. For example, while SSA is an IR that is widely recognized as enabling many efficient and powerful optimizations it cannot be easily applied in the presence of pointers. In this paper we present an intermediate representation, Pegasus, that enables efficient and powerful optimizations in the presence of pointers. Pegasus also increases available parallelism by supporting aggressive predication without giving up the features of SSA.

One of Pegasus' main design goals is to support spatial computation. Spatial computation refers to the direct execution of high-level language programs in hardware. Each operation in the program is implemented as a hardware operator. Data flows from one operation to another along wires that connect the operations.

Pegasus is a natural intermediate representation for spatial computation because its semantics are similar to that of an asynchronous circuit. Pegasus is an executable IR which unifies predication, static-single assignment, may-dependences, and data-flow semantics. The most powerful aspect of Pegasus is that it allows the essential information in a program to be represented directly. Thus, many optimizations are reduced to a local rewriting of the graph. This makes Pegasus a particularly scalable representation, yielding efficient compilation, even when compiling entire programs to circuits.

In this paper we describe optimizations which increase memory parallelism, eliminate redundant memory references, and increase the pipeline parallelism found in loops. We show how Pegasus' explicit representation of both control and data dependences allows these optimizations to be concisely expressed.

2 Example

We motivate our memory representation with the following example:

```
void f(unsigned*p, unsigned a[], int i)
{
    if (p) a[i] += *p;
    else   a[i] = 1;
    a[i] <<= a[i+1];
}
```

This program uses `a[i]` as a temporary variable. We have compiled this program using the highest optimization level using seven different compilers:

- gcc 3.2 for Intel P4, `-O3`,

- Sun WorkShop 6 update 2 for Sparc, `-xO5`,
- DEC CC V5.6-075 for alpha, `-O4`,
- MIPSpro Compiler Version 7.3.1.2m for SGI, `-O4`,
- SGI ORC version 1.0.0 for Itanium, `-O4`,
- IBM AIX cc version 3.6.6.0, `-O3`
- CASH.

Only CASH and the AIX compiler removed all the useless memory accesses made for the intermediate result stored in `a[i]` (two stores and one load). The other compilers retain the intermediate redundant stores to `a[i]`, which are immediately overwritten by the last store in the function. The CASH result is even more surprising in light of the simplicity of the analysis it performs: the optimizations consist only of reachability computations in a DAG and localized code transformations (term-rewriting). The point of this comparison is not to show the superiority of CASH; rather, we want to point out that optimizations which are complicated enough to perform in a traditional representation (so that most compilers forgo them), are very easy to express, implement and reason about when using a better program representation.

The strength of CASH originates from Pegasus, its internal program representation. The optimization occurs in three steps, detailed in Figure 1. (We describe Pegasus in Section 3 and details of these optimizations in Section 5.)

Figure 1A depicts a slightly simplified[1] form of the program representation before any optimizations. Each operation is represented by a node in a graph. Edges represent value flow between nodes. Pegasus uses predication [20]; throughout this paper, dotted lines represent predicate values, while dashed lines represent *may dependences* for memory access operations; these lines carry *token* values. Each memory operation has a predicate input, indicating whether it ought to be executed, and a token input, indicating that the side-effects of the prior dependent operations have occurred. At the beginning of the compilation process the token dependences track the original program control-flow. The nodes denoted by "V" represent operations that combine multiple input tokens into a single token for each output; these originally represent joins in the program control-flow. The "*" node is the token input, indicating that side-effects in the previous part of the program have completed.

CASH first proves that `a[i]` and `a[i+1]` access disjoint memory regions, and thus commute (nodes 3/5, 2/5 and 5/6). By removing the token edges between these memory operations the program is transformed as in Figure 1B. Notice that a new combine operator is inserted at the end of the program; its execution indicates that all prior program side-effects have occurred.

In Figure 1B, the load from `a[i]` labeled 4 immediately follows the two possible stores (2 and 3). No complicated dataflow or control-flow analysis is required to deduce this fact: it is indicated by the tokens flowing directly from the stores to the load. As such, the load can be removed and replaced with the value of the executed store. This replacement is shown in Figure 1C, where the load has been replaced by a *multiplexor* 7, drawn as a trapezoid. The multiplexor is controlled by the predicates of the two stores to `a[i]`, 2 and 3: the store that executes (i.e., has a "true" predicate at run-time), is the one that will forward the stored value through the multiplexor.

As a result of the load elimination, in Figure 1C the store 6 to `a[i]` immediately follows (and post-dominates) the other two stores, 2 and 3, to the same address; in consequence, 2 and 3 are useless, and can be completely removed, as dead code. Again, the post-dominance test does not involve any control-flow analysis: it follows immediately from the representation, because (1) the stores immediately follow each other, as indicated by the tokens connecting them; (2) they occur at the same address and (3) each predicate of an earlier store implies the predicate of the latter one, which is constant "true" (shown as 1), indicating the post-dominance. The post-dominance is determined by only elementary boolean manipulations. The transformation from C to D is accomplished in two steps: (a) the predicates of the prior stores are "and"-ed with the negation of the predicate of the latter store (i.e., the prior stores should occur only if the latter one doesn't overwrite them) and (b) stores with a constant "false" predicate are completely removed from the program as dead code.

2.1 Related Work

The explicit representation of memory dependences between program operations has been suggested numerous times in the literature; for example in Pingali's Dependence Flow Graph [21], or Steensgaard's adaptation to Value-Dependence Graphs [24]. As Steensgaard has observed, this type of representation is actually a generalization of SSA designed to handle value flow through memory. Other researchers have explored the use of SSA for handling memory dependences, e.g.,[7, 8, 14, 5, 6, 13].

The original contributions of this work are:

- We apply the memory access optimization techniques in the context of a *spatial compiler,* which translates C programs to hardware circuits.
- We show how to use predication and SSA together for memory code hoisting (subsuming partial redundancy elimination and global common-subexpression elimination for memory operation), removing loads after stores, dead store removal, and loop-invariant code motion for memory operations; our algorithms rely on boolean manipulation of the controlling predicates (Section 5).

[1] We have elided the address computation part.

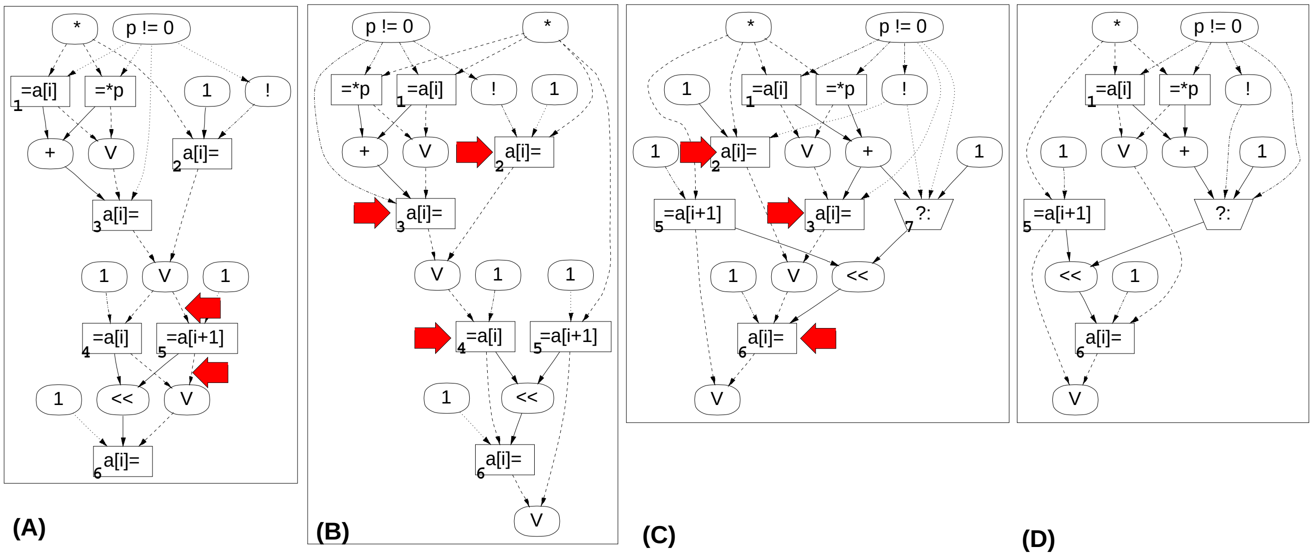


Figure 1: *Program transformations [dotted lines represent predicates, while dashed lines represent tokens]: (A) removal of two extraneous dependences between a[i] and a[i+1], (B) load from a[i] bypasses directly stored value(s), (C) store to a[i] kills prior stores to the same address (D) final result. This figure is automatically generated by the CASH compiler using the Graphviz[18] graph visualization tool.*

- We describe a representation and series of optimizations which expose pipeline parallelism in the memory accesses within loops (Section 6).
- We present (in Section 6.3) *loop decoupling*, a new loop parallelization algorithm, which decomposes a single loop into several independent loops and uses a novel computational primitive, a *token generator* to dynamically bound the slip between the newly created independent loops, preserving memory dependences at run-time.

3 Pegasus

In this section we describe Pegasus, our intermediate representation; we emphasize the representation of memory operations. Details about Pegasus, including a complete algorithm for building it from a control-flow graph representation, a formal semantics, and detailed descriptions of numerous scalar optimizations can be found in [1, 2].

3.1 From Control-Flow to Data-Flow

CASH essentially transforms the initial C program into a pure dataflow representation. All C control-flow constructs are transformed into equivalent dataflow operations, as described below. The resulting dataflow representation has semantics similar to that of an asynchronous circuit: data producers handshake with data consumers on data availability and data consumption. This explicit dataflow representation naturally exposes loop-level parallelism in the form of loop pipelining [22]. The representation is a directed graph where nodes are operations and edges indicate value flow. A node may fanout the data value it produces to multiple nodes; the fanout is represented as multiple graph edges, one for each destination.

Predication and speculation. In order to uncover more instruction-level parallelism, and to reduce the impact of control-flow dependences, aggressive predication and speculation are performed. We leverage the techniques used in compilers for predicated execution machines [17]: we collect multiple basic blocks into one hyperblock[2]; each hyperblock is transformed into straight-line code through the use of the predicated static single-assignment (PSSA) form [4]. All instructions without side-effects are aggressively predicate-promoted. Currently only static information is used to produce the hyperblocks (i.e., profiling is not used).

A difference between the algorithms we describe here and truly global ones is that those described in this paper are applied only within the scope of a single hyperblock.

Multiplexors. When multiple definitions of a value reach a join point in the control-flow graph, they must be merged together. This is done by using multiplexors[3]. The mux data inputs are the reaching definitions. We use *decoded muxes*, which select one of n reaching definitions of a variable; such a mux has $2n$ inputs: one data and one predicate input for each definition (see for example Fig-

[2] A hyperblock is a contiguous portion of the program control-flow graph, having a single entry point and possibly multiple exits.

[3] The multiplexors correspond directly to the ϕ functions in the SSA representation.

ure 1C). When a predicate evaluates to "true", the mux selects the corresponding input. The mux predicates correspond to the path predicates in PSSA.

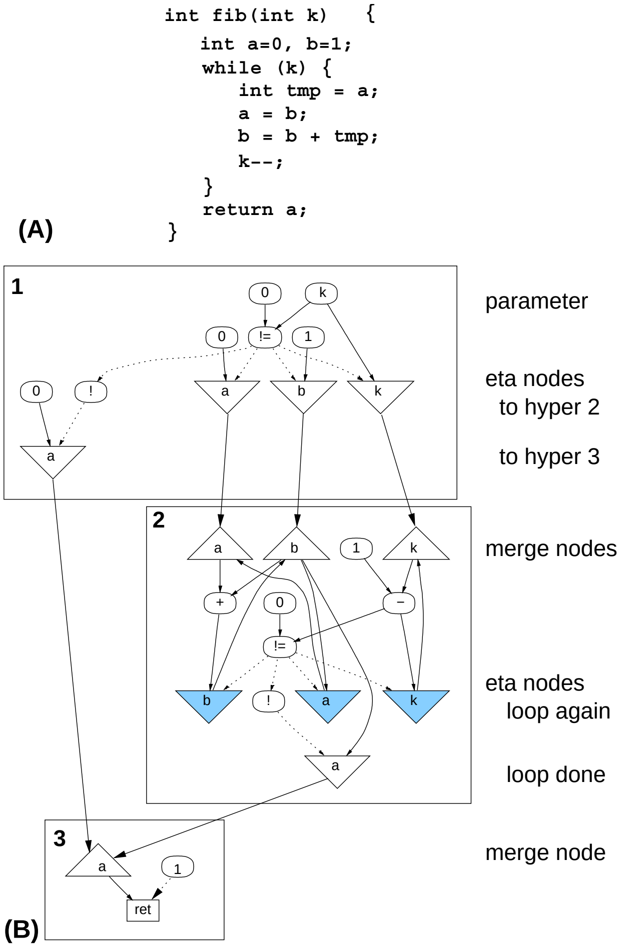
Merge and eta operators. After hyperblock construction, the only remaining control-flow constructs are inter-hyperblocks transfers, including loops. The next compilation phase stitches the hyperblocks together into a dataflow graph representing the whole procedure by creating dataflow edges connecting each hyperblock to its successors. Each variable live at the end of a hyperblock gives rise to an *eta* node. Eta nodes were originally introduced in the Gated-Single Assignment form [19]. Eta nodes have two inputs—a value and a predicate—and one output. When the predicate evaluates to "true," the input value is moved to the output; when the predicate evaluates to "false," the input value and the predicate are simply consumed, generating no output. A hyperblock with multiple predecessors receives control from one of several different points; such join points are represented by *merge* nodes.

Figure 2 illustrates the representation of a program comprising multiple hyperblocks, including a loop. The eta nodes (shown as triangles pointing down) in hyperblock 1 will steer data to either hyperblock 2 or 3, depending on the test `k!=0`. Note that the etas going to hyperblock 2 are controlled by this predicate, while the eta going to hyperblock 3 is controlled by the complement. There are merge nodes (shown as triangles pointing up) in hyperblocks 2 and 3. The ones in hyperblock 2 accept data either from hyperblock 1 or from the back-edges in hyperblock 2 itself. The arrows going up are back-edges denoting the flow of data along the `while` loop. The merge node in hyperblock 3 can accept control either from hyperblock 1 or from hyperblock 2.

Side-Effects Operations with side-effects are parameterized with a predicate input, which indicates whether the operation should take place. If the predicate is false, the operation is not executed and it generates a token instantaneously (loads and procedure calls generate an arbitrary result when their controlling predicate is false).

3.2 Tokens

Not all program data dependences are explicit: operations with side-effects can interfere with each other through memory. To ensure correctness, the compiler adds edges between operations whose side-effects may not commute. Such edges do not carry data values, instead, they carry an explicit synchronization *token*. Operations with side-effects wait for the presence of a token before executing; on completion, these instructions generate an output token to enable execution of their dependents. The token is really a data type with a single value, requiring 0 bits of information; in hardware it can be implemented with just the "data ready" signal. At run-time, once a memory operation has started execution and its effect is *guaranteed* to occur, the operation generates tokens for its dependents. Note that the token can be generated before memory has been updated.



```
int fib(int k)   {
   int a=0, b=1;
   while (k) {
      int tmp = a;
      a = b;
      b = b + tmp;
      k--;
   }
   return a;
}
```

Figure 2: *(A) Iterative C program computing the k-th Fibonacci number and (B) its Pegasus representation, comprised of 3 hyperblocks. The dotted lines represent predicate values.*

Operations with memory side-effects (loads, store, calls and returns) all have a token input. When a side-effect operation depends on multiple other operations (e.g. a write operation following a series of reads), it must collect one token from each of them. For this purpose a *combine* operator is used; a combine has multiple token inputs and a single token output; the output is generated after it receives all its inputs. Figure 3a shows a token combine operator, depicted by "V"; dotted lines indicate token flow. From now on, to reduce clutter, we elide combines without ambiguity, depicting Figure 3a as in Figure 3b.

Token edges explicitly encode data flow through memory. In fact, the token network can be interpreted as an SSA

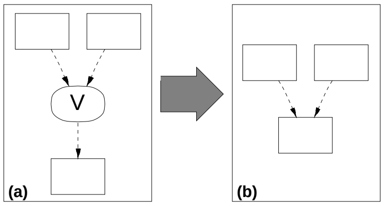


Figure 3: *(a) "Token combine" operations emit a token only on receipt of tokens on each input. (b) Reduced representation of the same circuit, eliding the token combine.*

```
extern int a[], b[];

void g(int *p, int i)
{
    b[i+2] = i & 0xf;
   a[i] = b[i] + *p;
}
```

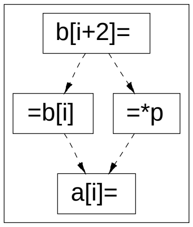


Figure 4: *A program and the schematic Implementation of its memory synchronization, assuming all memory instructions can access any memory word (i.e. with no pointer analysis).*

form encoding of the memory values, where combine operations are similar to ϕ-functions. The tokens encode both true-, output- and anti-dependences, and they are "may" dependences.

3.3 Synchronization-Insertion Algorithm

During the construction of the Pegasus graph token edges are inserted using the following algorithm:

- A flow-insensitive analysis determines local scalar values whose address is never taken; these are assigned to registers. In Figure 2 values `a`, `b` and `k` are scalars. All other data values (non-scalars, globals or scalars whose address is taken) are manipulated by load and store operations through pointers. These are the subject of this paper.
- Each memory access operation has an associated read/write set [10] (also called "tags" in [16] or M-lists in [6]): the set contains the memory locations that the operation *may* access.
- Memory instructions are created by traversing the control-flow graph in program order.
- Instruction i receives tokens from all instructions j such that there is a control-flow path from j to i, and i does not commute with j (i.e. i and j aren't both memory reads) and the read/write sets of i and j overlap.

Figure 4 shows a sample program and the constructed representation. Notice that the two memory reads (of `b[i]` and `*p`) have not been sequentialized, since read operations always commute.

3.4 Transitive reduction of the token graph

The graph formed by token edges is maintained in a transitively reduced form by the compiler throughout the optimization phases. This is required for the correctness of some of the optimizations described below. The presence of a token edge between two memory operations in a transitively reduced graph implies the following properties:

- the two memory operations may access the same memory location, *and*
- there is no intervening memory operation affecting that location.

4 Increasing Memory Parallelism

CASH performs a series of optimizations with the goal of increasing the available memory parallelism and reducing the number of redundant memory operations. The basic idea is to remove token edges between memory operations that are actually independent.

4.1 Removing dead memory operations

As mentioned, memory operations are predicated and should execute only when the run-time value of the predicate is "true". In particular, a side-effect operation having a constant "false" predicate can be completely removed by the compiler from the program; its token input is connected to its token output. Such constant false predicates can arise due to control-flow optimizations (such as `if` statements with constant tests) or due to redundant memory optimizations, described below.

4.2 Immutable objects

Accesses made through pointers to constants (such as constant strings or pointers declared `const`) can be optimized: if the pointed value is statically known, the pointer access can be removed completely; otherwise, the access does not require *any* token, since it does not need to be serialized with any other operation. The access also does not need to generate a token.

4.3 Removing unnecessary token edges

CASH processes token edges by considering pairs of instructions directly synchronized (i.e. one produces a token consumed by the other) and tries to prove that they will never simultaneously access the same memory address. If this proof succeeds, the token between these instructions is removed. The transitive closure of the token graph (minus the removed edge) must be preserved, so new tokens are inserted for this purpose. The basic algorithm step is illustrated in Figure 5. (Notice that, although the number of explicit synchronization tokens may increase, the total number of synchronized pairs in the transitive closure of the token graph is always reduced as a result of token removal.)

CASH uses several heuristics to prove that some addresses can never be equal: (1) symbolic computation is

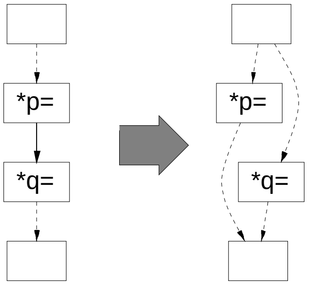


Figure 5: *Removing unnecessary tokens when heuristics can prove that* p *and* q *do not alias.*

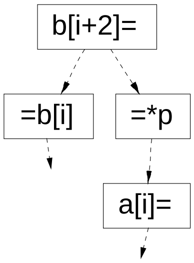


Figure 6: *Implementation of the sample program in Figure 4 when using read/write sets information. Since arrays* a *and* b *are disjoint, there is no need for a token edge between accesses to their elements.*

used to check whether two address expressions are always different (as shown in Figure 1A-B) (2) within loops induction variable analysis is used to prove that induction variables with the same induction step and different starting values are always different and (3) pointer analysis is used to detect instructions having disjoint read/write sets (see Figure 6 for an example).

5 Removing redundant memory accesses

In this section we describe several optimizations which take advantage of the SSA-like nature of our representation for memory in order to discover useless memory accesses. The explicit token representation makes the following optimizations easy to describe and implement for memory operations: load and store merging (subsuming partial redundancy elimination and global common-subexpression elimination), dead store removal, load-after-store removal and loop-invariant code motion. Token edges simultaneously represent control-flow and dependence information, which makes it easy for the compiler to focus its optimizations on interfering memory operations. The first three transformations described are implemented simply as term-rewriting rules, much as depicted in the associated figures.

Optimizations involving memory operations must properly handle the controlling predicates: an optimization should not create cycles in the program representation. For example, when merging two loads, a cycle may be created if the predicate of one of the loads is a function of the result of the other load. Testing for the cycle-free condition is easily accomplished with a reachability computation in the Pegasus DAG which ignores the back-edges. By caching the results of the reachability computation for a batch of optimizations, its amortized cost remains linear.

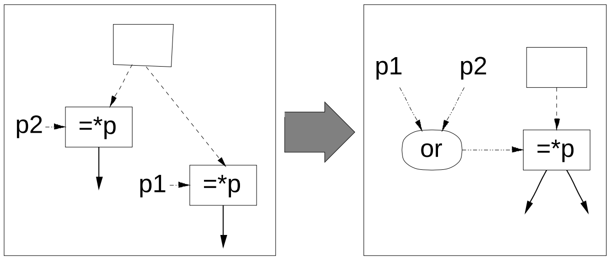


Figure 7: *Load operations from the same address and having the* same token source *can be coalesced; the predicate of the resulting load is the disjunction of the original predicates.*

Notice that the optimizations described in this section all assume aligned memory accesses. All are applicable only when the optimized memory operations access the same address and manipulate the same amount of data (i.e. they do not handle the case of a store word followed by a load byte).

5.1 Merging equivalent memory operations

We first present an optimization merging distinct but equivalent memory operations. It generalizes global common-subexpression elimination, partial redundancy elimination and code hoisting for memory accesses, by merging multiple accesses to the same memory address into one single operation. How this optimization operates for loads is shown in Figure 7; this optimization is applicable to stores as well. If the two predicates of the loads are the same node, this optimization becomes common-subexpression elimination for memory operations.

5.2 Redundant store removal

Figure 8 depicts the store-before-store code transformation (used in the example in Figure 1C). We ensure that at run time the first store is executed only if the second one is not, since the second one will overwrite the result. If the predicate of the prior stores implies the predicate of the following store (which happens if the second post-dominates the first), the first predicate will become constant “false”. If the boolean manipulation performed by the compiler can prove this fact, the first store can be eliminated using the rule from Section 4.1. Notice that, if the predicate is false, but the boolean manipulation cannot simplify it, the store operation nevertheless does not occur at run-time.

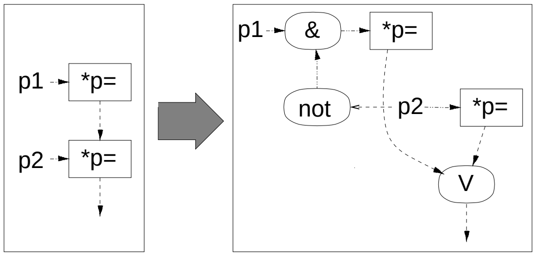


Figure 8: *Redundant store-after-store removal: a store immediately preceding another store having the same address should execute only if the second one does not. Since the two stores may never occur at the same time, the token edge connecting them can be safely removed.*

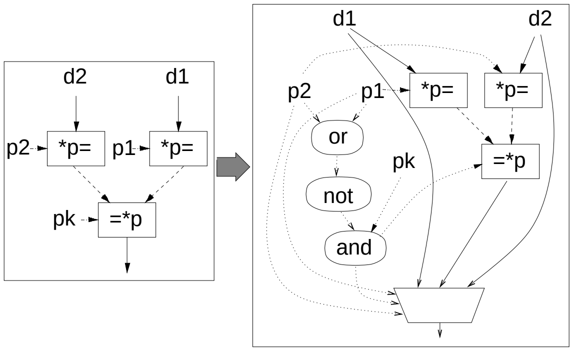


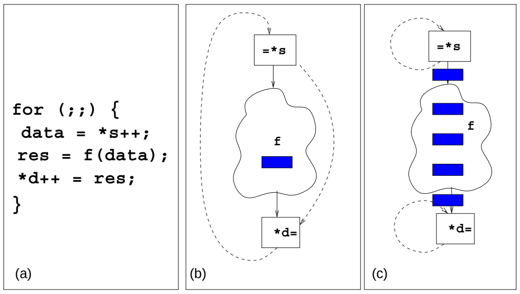
Figure 9: *Removal of loads following stores from the same address. The stored value is directly forwarded if any of the stores occurs, otherwise the load has to be performed.*

5.3 Load after store removal

In Figure 9 we show how loads following a store can directly bypass the stored value (Figure 1B shows an instance of this transformation). This transformation ensures that if either of the stores is executed (i.e. its predicate is “true”), then the load is not executed. Instead, the stored value is directly forwarded. The load is executed only when none of the stores takes place. If the stores collectively dominate the load (as formally defined by Gupta in [11]), the latter is completely removed, since the corresponding multiplexor predicate will become constant “false”.

5.4 Loop-invariant load motion

A scalar computation is loop invariant if: (1) it is constant, (2) its value circulates around the loop unchanged through a merge-eta loop or (3) it has no side-effect and all its inputs are loop-invariant. Loop-invariant code motion for memory operation does not require any special handling: the same algorithm which handles loop-invariant code motion for scalars works equally well. The requirement for a memory operation to be loop-invariant is that all its inputs are loop-invariant, including the predicate and the token. Then such an operation (together with its loop-invariant inputs) can be lifted to a newly created, loop-header hyperblock, preceding the loop. The next section describes how loops handle tokens.



```
for (;;) {
 data = *s++;
 res = f(data);
 *d++ = res;
}
```

Figure 10: *Fine-grained memory synchronization facilitates loop pipelining.*

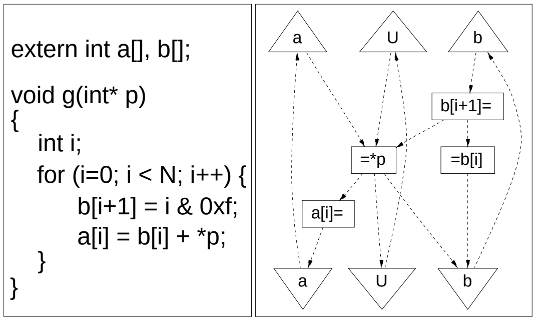
Notice that loop-invariant stores are not detected using this scheme, since they generate a fresh token in each iteration; thus, their token input is not loop invariant.

6 Pipelining loops with fine-grained synchronization

The algorithms presented in this paper, and in particular in this section, are motivated by our intended target: direct hardware implementation of the program. Pipelining loops is especially important to fully realize the performance of dataflow machines, synthesized directly in hardware. The optimizations presented in this section increase the opportunities for pipelining by removing dependencies between memory operations from different loop iterations and by creating structures which allow them to execute in parallel.

The key to increasing the amount of pipeline parallelism available is to correctly handle memory synchronization across loop iterations. We do this by carrying fine-grained synchronization information across loop iterations using several merge-eta token loops. Figure 10 illustrates schematically how pipelining is enabled by fine-grained synchronization. In 10(a) is a schematic code segment which reads a source array, computes on the data and writes the data into a distinct destination array. Figure 10(b) shows a traditional implementation which executes the memory operations in program order: the reads and writes are interleaved. Meanwhile, Figure 10(c) shows how separately synchronizing the accesses to the source and destination array “splits” the loop into two separate loops which can advance at different rates. The “producer” loop can be several iterations ahead of the “consumer” loop, filling the computation pipeline.

Figure 11 shows an example loop and its representation (we show only the memory operations and their synchronization). Notice that each merge-eta circuit is associated with a particular set of memory locations. There are cir-



```
extern int a[], b[];

void g(int* p)
{
    int i;
    for (i=0; i < N; i++) {
        b[i+1] = i & 0xf;
        a[i] = b[i] + *p;
    }
}
```

Figure 11: *Implementation of memory synchronization in loops.*

cuits for `a` and `b` and one, labeled `U`, which represents all anonymous memory objects. The merge node tagged with `a` represents all accesses made to `a` in the program *before* the current loop iteration, while the eta represents the accesses *following* the current loop iteration.

The circuit was generated using the following algorithm:

- The read-write sets of all pointers in the loop body are analyzed; the objects in these sets are grouped into equivalence classes: two objects are equivalent if they appear together in all read-write sets. For the example in Figure 11, the read-write sets are as follows:

`a`→ {`a[]`}	pointer `a` points to array `a[]`
`b`→ {`b[]`}	
`p`→ {`a[],b[],U`}	`p` can point to anything

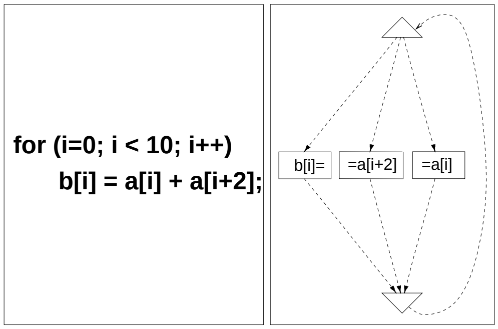
The objects in the read-write sets are `a[]`, `b[]` and `U`. In this example each object is in its own equivalence class.

- A merge-eta cycle is created for each equivalence class.
- The merge and eta are treated as writes to all objects in the equivalence class.
- The loop body is synthesized as straight-line code, as if preceded by all merges and succeeded by all etas.

6.1 Read-only accesses

As mentioned above, the token-insertion algorithm for building loops treats the merge and eta operations as writes to the corresponding object(s) when building the token edges. Treating them as writes ensures that all operations accessing the object in the i-th iteration will complete before the $i+1$-th iteration can start, preventing pipelining. Sometimes this amount of synchronization is excessive. In particular, we can optimize the loop when all operations are reads: we can then safely overlap different loop iterations by pipelining.

If a memory object accessed in a loop does not appear in any of the write-sets, the loops is optimized by splitting it into *three* parallel loops, enabling fast access to all read-only objects: (1) one loop containing the accesses to all the written objects (2) a *generator* loop, which generates tokens to enable the read operations for all the loop iterations, allowing operations from multiple iterations to issue simultaneously, and (3) a *collector* loop, which collects the tokens from the read operations in all iterations; this ensures that the loop terminates only when all reads in all iterations have occurred.



```
for (i=0; i < 10; i++)
    b[i] = a[i] + a[i+2];
```

Figure 12: *Sample code and naive implementation of a loop containing read-only accesses to array* `a`*.*

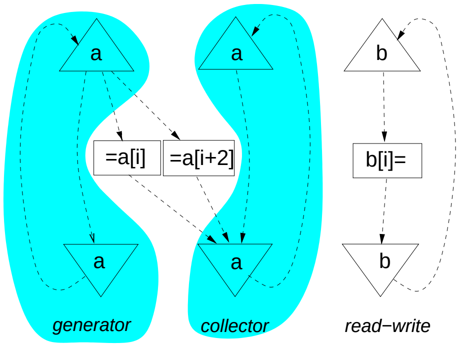


Figure 13: *Optimized implementation of the read-only loop from Figure 12.*

An example program and implementation of the token-insertion algorithm for loops is given in Figure 12; applying the read-only optimization results in Figure 13.

6.2 Using address monotonicity

There is no need to synchronization writes to different addresses. For example, all writes to `b` in Figure 12 are to different addresses. The compiler attempts to discover such writes, where the addresses vary strictly monotonically from one iteration to the next using an extended form of induction variable analysis as in Wolfe's [25]. When such a case is found, as in Figure 13, the loop is transformed as described above for read-only accesses, causing multiple loop iterations to be initiated in pipelined fashion. The result after the optimization of the read-write loop of `b` from Figure 13 is shown in Figure 14.

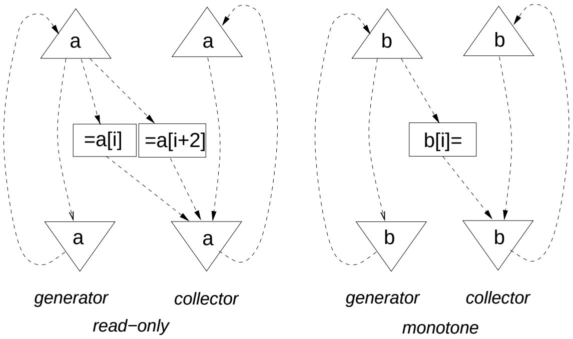
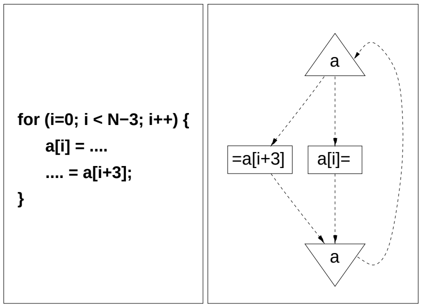


Figure 14: *The loop in Figure 13 after simplifying the* `b` *loop using address monotonicity.*



```
for (i=0; i < N−3; i++) {
      a[i] = ....
      .... = a[i+3];
}
```

Figure 15: *Program and its implementation before loop decoupling.*

6.3 Loop Decoupling: Using Dependence Distances

The most sophisticated pipeline-enabling optimization in our compiler is a novel form of loop parallelization which is particularly effective for spatial compilation. This transformation, *loop decoupling* is remotely related to loop skewing and loop splitting. Analogously to the optimization for read-only values, it "vertically" slices a loop into multiple independent loops, which are allowed to *slip* with respect to each other (i.e. one can get many iterations ahead of the other). The maximum slip is derived from the *dependence distance*, and ensures that the loops slipping ahead do not violate any program dependence. To dynamically control the amount of slip a new operator is used, a *token generator*.

We illustrate its effect on the circuit in Figure 15, obtained after all previously described optimizations are applied to the shown program. Dependence analysis indicates that the two memory access are at a fixed distance, of 3 iterations. The two accesses are separated into two independent loops by creating a token for each of them. The `a[i+3]` loop can execute as quickly as possible. To preserve correctness, the loop for `a[i]` can only execute 3 iterations ahead of the `a[i+3]` loop, because otherwise it would violate the dependences. In order to dynamically bound the slip between these two loops a token generator operation which can generate 3 tokens is inserted between the two loops. The resulting structure is depicted in Figure 16. The token generator can instantly generate 3 tokens; subsequently it generates additional tokens upon receipt of tokens at the input.

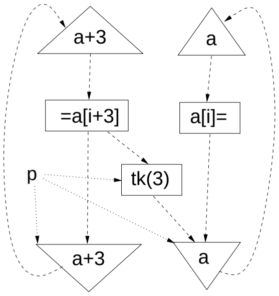


Figure 16: *The program from Figure 15 after loop decoupling. We have also depicted p, the loop-controlling predicate.*

More precisely, a token generator **tk**(n) has two inputs: a predicate and a token, and one output, a token. n is a positive integer, the dependence distance between the two accesses. The token generator maintains a counter, initialized with n. The predicate input is the loop-controlling predicate. On receipt of a "true" predicate, the generator decrements the counter, and if the counter is positive, it emits a token. On receipt of a token, it increments the counter. On receipt of a "false" predicate (indicating completion of the loop), the counter is reset to n (3 in our example).

The `a[i]` loop must receive tokens from the `a[i+3]` loop in order to make progress. It may receive the first three tokens before any iteration of the latter has completed, directly from the token generator, so it can slip up to three iterations ahead. Afterwards, it will receive a new token enabling it to make progress only after the `a[i+3]` loop completes one iteration. In contrast, the `a[i+3]` loop can slip an arbitrary number of iterations ahead (the token generator stores the extra tokens in its internal counter).

Notice that after loop decoupling each of the resulting loops contains only monotone induction variables, so it can be further optimized as described in Section 6.2. The final result is shown in Figure 17.

The idea of explicitly representing the dependence distances in a program can be found in Johnson's et al. paper "An executable representation of distance" [12]. While that paper gives a general scheme for representing dependence distances in nested loops as functions of multiple loop indices, it does not discuss how the resulting representation can be used to optimize the program. Our scheme can also be generalized to deal with nested loops, although our cur-

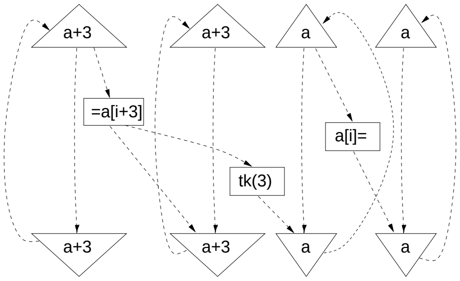


Figure 17: *Program after loop decoupling and monotone induction variable optimization.*

Optimization	**LOC**
Useless dependence removal	160
Immutable loads	70
Dead-code elimination (incl. memory op)	66
Load-after-load and store-after-store removal	153
Redundant load and store removal (PRE)	94
Transitive reduction of token edges	61
Loop-invariant code discovery (scalar and memory)	74
Loop decoupling+monotone loops	310

Table 1: *Lines of C++ code, including comments and white-space, implementing the optimizations described in this paper.*

	Kernels			Pragmas	
Benchmark	Funcs	Lines	Cov	#	Cov
adpcm_e	1	93	100	6	100
adpcm_d	1	70	100	5	100
gsm_e	76	1667	97	20	43
gsm_d	75	1644	96	15	71
epic_e	32	629	12	11	84
epic_d	13	529	77	3	100
mpeg2_e	94	4766	96	1	72
mpeg2_d	112	4942	99	1	20
jpeg_e	108	3268	30	7	40
jpeg_d	100	3203	82	8	69
pegwit_e	95	1556	94	0	
pegwit_d	95	1556	100	1	30
g721_e	15	386	97	0	
g721_d	20	463	96	0	
mesa	225	9892	61	6	33
099.go	371	12259	76	4	1
124.m88ksim	104	2379	70	17	28
129.compress	17	590	98	4	9
130.li	35	300	60	2	5
132.ijpeg	120	3404	70	0	
134.perl	220	8505	65	0	
147.vortex	241	7075	60	0	
Total	2170	69176	n/a	111	n/a

Table 2: *Statistics of the program fragments compiled and number of `pragma` statements introduced. Numeric columns are: total number of functions compiled, total source-level lines in these functions and dynamic program coverage in these functions; number of `pragma independent` statements inserted, percent of the program running time in the modified functions.*

rent implementation only handles innermost loops. The incarnation of the dependence distance in the "token generator" operator, which is used to dynamically control the slip between decoupled loop iterations is a new idea, which, we believe, is indeed an *executable representation of distance*.

7 Results

In this section we evaluate the CASH compiler on programs from the Mediabench [15] and SpecInt95 [23] benchmark suites. We present results for all programs which successfully went through our infrastructure.

7.1 Compilation process measurements

One measure of the Pegasus' efficiency is in the amount of code it takes to implement each of the optimizations which, as shown in Table 1, is quite small.

Limitations in our current simulation infrastructure prevented us from successfully evaluating all functions in the programs. The left side of table 2 shows the amount of code we have compiled and evaluated from each benchmark, measured according to three different criteria: number of functions, source-code lines and run-time (as measured on a SimpleScalar [3] 4-way out-of-order processor). All measurements reported in this paper are only for these selected functions.

CASH is notably slower than gcc (version 3.2 running -O3), up to 30 times for perl, on average being about 10 times slower. About half the time spent in CASH is spent on the optimizations described in this paper and the following optimizations: constant folding, global constant propagation, loop unrolling, strength reduction, dead-code elimination, loop-invariant code motion, partial redundancy elimination, global common-subexpression elimination, unreachable code elimination, re-association, algebraic simplifications and intra-procedural pointer analysis. The remaining time is spent doing parsing, I/O, pointer analysis and in constructing the hyperblocks.

Approximating inter-procedural pointer-analysis. In a language like C which permits the liberal use of pointers a pointer analysis made only at the procedure level must be exceedingly conservative. In order to approximate a more precise whole-program analysis with little effort we resorted to manual annotation of the program.

`#pragma independent p q` guarantees to the compiler that, in the current scope, pointers `p` and `q` never point to the same address. This pragma is somewhat sim-

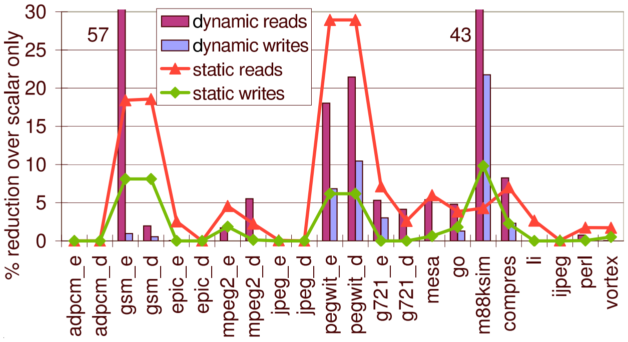


Figure 18: *Static and dynamic reduction of memory traffic in percents. The baseline is obtained with CASH and only scalar optimizations (all memory optimizations turned off). The lines represent static counts, while the bars are dynamic counts.*

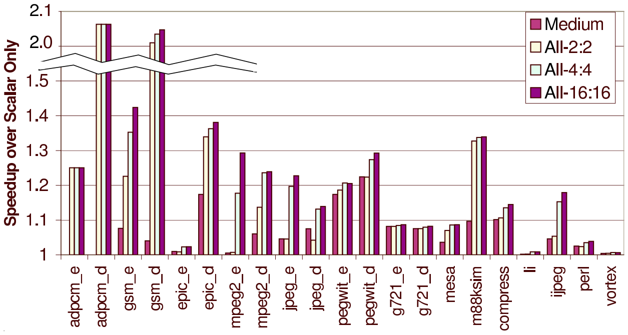


Figure 19: *Program speed-up over scalar-only optimizations when varying compiler optimizations and memory bandwidth. The first two bars compare different optimizations for dual-ported memory system. The last two bars show the effect of increasing the memory bandwidth for the highest level of optimizations.*

ilar to the proposed C99 `restrict` keyword, but much more intuitive to use. We use a simple, intraprocedural version of connection analysis [9] to propagate the independence information through pointer expressions. If two pointer references are declared independent, a token is not necessary to synchronize them. We have profiled the programs and inserted annotations in the most important functions. A pragma is most often used to indicate that the pointer arguments to a function do not alias to each other. Only a handful of pragmas were inserted into the programs as shown in the right-hand side of Table 2. For a few programs these pragmas are extremely effective in aiding optimization.

7.2 Static measurements

The space complexity of the internal representation when doing no memory-related optimizations is asymptotically the same as a classical SSA representation. There was some concern that the fine-grained representations of tokens would blow up quadratically. However, independent of which memory optimizations were turned on or off, the size of the IR never varied by more than 3%.

The number of memory operations varies more substantially; up to 8% of the static stores and up to 28% of the loads are removed. The line graphs in Figure 18 quantify these effects for all the programs.

Notice we can't directly compare these metrics with compilers such as gcc, since in hardware CASH can allocate an unbounded number of registers, while gcc must sometimes spill registers to memory, increasing memory traffic.

7.3 Dynamic measurements

We evaluated the effectiveness of our optimizations on a coarse hardware simulator. We assume that the entire program can be translated to hardware. Each operation has the same latency as in a pisa architecture SimpleScalar simulator. All memory operations inject requests into a load-store queue with a finite number of ports and a finite size. We have evaluated several memory systems, ranging from perfect memory to a realistic memory system with two levels of cache. Spatial computation does not need to issue instructions, so there's no instruction cache. The L1 cache has 2 cycles hit latency and 8kb, while the L2 cache has 8 cycles hit latency and 256kb. Memory latency is 72 cycles, with 4 cycles between consecutive words. The memory is dual-ported. The data TLB has 64 pages with a 30 cycle TLB miss cost.

There are three ways in which our algorithms can increase the performance of a program:

- By reducing the number of memory operations executed;
- By overlapping many memory operations (increasing traffic bandwidth) through loop pipelining;
- By creating more compact schedules and issuing the memory operations faster.

The bars in Figure 18 shows that the compiler reduces the dynamic amount of memory references for some of the programs. This arises from the algorithms described in Section 5. However, the latter two ways of increasing performance appear to be more important as one compares Figure 18 to Figure 19. By comparing the rightmost three bars of Figure 19 we can also see that although performance improves with an increase in memory bandwidth, even small amounts of bandwidth can be utilized quite effectively by the compiler.

In addition to these overall statistics, we compiled each of the benchmarks with different sets of optimizations and simulated them with different memory systems. We found that the programs benefited most from using pointer analysis to reduce token edges during construction (Section 3.3), eliminating synchronization edges by disambiguating memory addresses (Section 4.3), and increasing pipelining from induction variable analysis (Section 6.2). This set of optimizations is shown by the first bar, labeled “Medium,” in Figure 19. In all but a few cases, the independence pragmas were helpful in augmenting the pointer analysis. The read-only optimizations in Section 6.1 were almost always not very profitable. Loop Decoupling (Section 6.3) was also not very useful as it was applicable in only 28 loops from all the programs. We expect that it would be more applicable to Fortran-type loops. We found that the result of applying optimizations together was more powerful than simply the product of their individual effect.

8 Conclusions

In this paper we have shown how powerful sophisticated memory optimizations can be simply implemented in our intermediate representation, Pegasus. The power of Pegasus is a direct result of its ability to explicitly represent all the dependences in a program—scalar dependences, may dependences through memory, and dependence distances around loops. This combines well with Pegasus’ use of a predicated SSA-like representation. We have shown how this representation can be used to increase memory parallelism, remove redundant memory operations, and increase the amount of pipeline parallelism available. Pegasus, and the optimizations presented here, is motivated by our desire to compile high-level language programs directly into hardware. However, many of the techniques used in CASH are applicable for traditional and parallel compilation.

Acknowledgements

We thank Girish Venkataramani, Dan Vogel and David Koes for discussions and help with pragma insertion, scripting and comments on previous versions of this manuscript. The comments of the anonymous reviewers helped enormously the presentation and pinpointed important errors in the presentation. This research is funded in part by the National Science Foundation under Grant No. CCR-9876248 and by Darpa under contract #MDA972-01-03-0005.

References

[1] M .Budiu and S. C .Goldstein. In *Proceedings of the 12th International Conference on Field Programmable Logic and Applications*, Montpellier (La Grande-Motte), France, September 2002.

[2] M .Budiu and S. C .Goldstein. Pegasus: An efficient intermediate representation. Technical Report CMU-CS-02-107, Carnegie Mellon University, May 2002.

[3] D .Burger and T. M .Austin. In *Computer Architecture News*, volume 25 (3), pages 13–25. ACM SIGARCH, June 1997.

[4] L .Carter, B .Simon, B .Calder, L .Carter, and J .Ferrante. *International Journal of Parallel Programming, special issue*, 28(6), 2000.

[5] F .Chow, R .Lo, S.-M .Liu, S .Chan, and M .Streich. In *Proc. of 6th Int’l Conf. on Compiler Construction*, pages 253–257, April 1996.

[6] K. D .Cooper and L .Xu. In *Workshop on Memory Systems Performance (MSP ’02)*, Berlin, Germany, June 2002.

[7] R .Cytron and R .Gershbein. In *Proceedings of the conference on Programming Language Design and Implementation (PLDI)*, pages 36–45. ACM Press, 1993.

[8] D. M .Gallagher. *Memory Disambiguation to Facilitate Instruction-Level Parallelism Compilation*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[9] R .Ghiya and L. J .Hendren. *International Journal of Parallel Programming*, 24 (6):547–578, 1996.

[10] R .Ghiya and L. J .Hendren. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 121–133, San Diego, California, January 1998.

[11] R .Gupta. In *ACM SIGPLAN-SIGACT 19th Symposium on Principles of Programming Languages*, pages 246–257, Albuquerque, New Mexico, January 1992.

[12] R .Johnson, W .Li, and K .Pingali. *Languages and Compilers for Parallel Computers*, 4, 1991.

[13] R .Kennedy, S .Chan, S.-M .Liub, R .Lo, P .Tu, and F .Chow. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):627 – 676, May 1999.

[14] C .Lapkowski and L. J .Hendren. In Springer, editor, *Proceedings of the 1998 International Conference on Compiler Construction*, volume 1383 of *LNCS*, pages 128–143, March 1998.

[15] C .Lee, M .Potkonjak, and W. H .Mangione-Smith. In *Micro-30, 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, 1997.

[16] J .Lu and K. D .Cooper. In *Proceedings of the 1997 ACM SIGPLAN conference on Programming language design and implementation*, pages 308–319. ACM Press, 1997.

[17] S. A .Mahlke, D. C .Lin, W. Y .Chen, R. E .Hank, and R. A .Bringmann. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, Dec 1992.

[18] A .open graph visualization system and its applications to software engineering. *Software Practice And Experience*, 1(5), 1999. http://www.research.att.com/sw/tools/graphviz.

[19] K. J .Ottenstein, R. A .Ballance, and A. B .Maccabe. In *Proceedings of the Conference on Programming Language Design and Implementation PLDI 1990*, pages 257–271, 1990.

[20] J. C. H .Park and M. S .Schlansker. On predicated execution. Technical Report HPL-91-58, Hewlett-Packard Laboratories, May 30 1991.

[21] K .Pingali, M .Beck, R .Johnson, M .Moudgill, and P .Stodghill. In *Proceedings of Principles of Programming Languages POPL*, volume 18, 1991.

[22] J .Silc, B .Robic, and T .Ungerer. *Parallel and Distributed Computing Practices*, 1 (1):3–30, 1998.

[23] Standard Performance Evaluation Corp. *SPEC CPU95 Benchmark Suite*, 1995.

[24] B .Steensgaard. In *In Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations*, pages 62–70, 1995.

[25] M .Wolfe. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, volume 27 (7), pages 162–174, New York, NY, 1992. ACM Press.