UC Irvine

ICS Technical Reports

Title

Mutation scheduling : a unified approach to compiling for fine-grain parallelism

Permalink

<https://escholarship.org/uc/item/2mf1p1tq>

Authors

Novack, Steven
Nicolau, Alexandru

Publication Date

1994-05-17

Peer reviewed

SLBAR

Z

699

C3

no. 94-23

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Mutation Scheduling: A Unified Approach
to Compiling for Fine-Grain Parallelism*

TECHNICAL REPORT 94-23

Steven Novack and Alexandru Nicolau
Department of Information and Computer Science
University of California
Irvine, CA 92717

---

*This work was supported in part by NSF grant CCR8704367 and ONR grant N0001486K0215.

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Mutation Scheduling: A Unified Approach to Compiling for Fine-Grain Parallelism*

TECHNICAL REPORT 94-23

Steven Novack and Alexandru Nicolau
Department of Information and Computer Science
University of California
Irvine, CA 92717

May 17, 1994

1 Introduction

In this paper we present Mutation Scheduling (MS), a unified, compiler-based approach for exploiting the functional, register, and memory bandwidth characteristics of arbitrary single-threaded fine-grain parallel architectures, such as VLIW, super-scalar, and super-pipelined. MS is a “value-oriented” approach to instruction scheduling that allows the computation of any given value to change dynamically during scheduling to conform to varying resource constraints and availability.

Generating code for single-threaded fine-grain parallel architectures consists of three phases: code selection, register allocation, and instruction scheduling. Code selection refers to deciding which sequence of machine operations will be used to compute each value needed by the program. Register allocation determines when these values will reside in the registers and when they will be transferred between memory and the register file. Instruction scheduling refers to mapping the selected operations to the appropriate functional units of the architecture so that the order of execution minimizes execution time while preserving the semantics of the original program. Conventional compilers perform these phases separately so that decisions made during code selection and register allocation can unnecessarily constrain the ability of the scheduling phase to utilize available machine resources. This problem exists even for conventional pipelined architectures, but becomes more critical for fine-grain parallel machines. For example, consider a super-scalar architecture consisting of three functional units, an adder, a shifter and a multiplier. If we want to generate the value $Y \leftarrow X * 2$, three possible single operation code selections are immediately obvious: “(add Y X X)”, “(mul Y X 2)”, and “(lshift Y X 1)”. The code selection phase would generally associate a cost, such as the sum of operation latencies, with each possible code sequence, and choose the one with the smallest cost. Of these three single-operation sequences, “(lshift

---

*This work was supported in part by NSF grant CCR8704367 and ONR grant N0001486K0215.

"Y X 1)" would generally be selected as the quickest (e.g. by a reduction in strength optimization pass), thus forcing the instruction scheduling phase to assign the operation to the shifter unit. However, when scheduling "(lshift Y X 1)" it might be the case that the shift unit is already committed to computing other operations (*at that point in the schedule*), while one or both of the other two units remains simultaneously idle. In this context, even though the shift operation has the lowest code selection cost, one of the other operations may have been more appropriate. Similarly, if registers are allocated prior to scheduling, spurious dependencies, such as write-after-write and write-after-read, are created that can prevent operations from being scheduled in parallel.¹

The Mutation Scheduling (MS) approach presented in this paper is based on integrating code selection and register allocation into the instruction scheduling phase in an attempt to "adapt" a given program to best fit the physical characteristics of the target architecture. MS works by associating each value, VAL, defined in the program with a set, called MUTATIONS(VAL), of functionally equivalent expressions, each of which computes VAL using some different resources of the target architecture.² At any given time during parallelization, exactly one of the expressions in MUTATIONS(VAL) will be used in the program graph to compute VAL. When scheduling the expression, EXPR, that currently generates VAL, if the resources for EXPR are unavailable, then another expression, or *mutation*, of VAL that better fits the resources may be substituted for it from MUTATIONS(VAL). When this happens, we say that VAL has been *mutated*.

MUTATIONS sets are also used for integrating register allocation into the scheduling process by allowing the MUTATIONS sets to change dynamically during scheduling to contain new expressions that may become available for a value. When a value, VAL, has already been computed and resides in a register, a reference to that register will become one of the expressions in MUTATIONS(VAL).³ If VAL is spilled to its "home" location in memory, then the expression "(load VAL home(VAL))" is also added to MUTATIONS(VAL). If at some point during scheduling, VAL is needed by some other expression, an evaluation function is applied to choose from MUTATIONS(VAL) the "best" way of accessing, or if necessary re-computing, VAL. If a register reference for VAL is still in MUTATIONS(VAL), then that register can simply be used. Otherwise, VAL must be "re-generated" by scheduling some expression from MUTATIONS(VAL). To allow for register spilling we will always insist that the two operation expression "(STORE home(VAL) VAL), (LOAD VAL home(VAL))"⁴ be a member of MUTATIONS(VAL). In addition, if VAL has already been spilled to memory then MUTATIONS(VAL) will also contain the single operation expression "(load VAL home(VAL))". Finally, MUTATIONS(VAL) may also contain other expressions that are capable of re-computing VAL from data already stored in the register file.

¹ Although, some compilers[NE90, NPW91, ME92, NN92b] do partially mitigate the effect of early register allocation by removing spurious dependencies using dynamic register renaming[CF87]. Others [RG82] perform a potentially very expensive post-scheduling register re-allocation.

² For example, if SHIFT and ADD operations are performed by the same functional unit and have the same definitions and uses, there would be no point in keeping both in the same MUTATIONS set.

³ Actually, a register reference is indicated by a NULL expression which indicates that no further computation is necessary for VAL since it already resides in the register file.

⁴ If this expression is selected as a new mutation for VAL, then it will be instantiated in its entirety, but nevertheless the STORE and LOAD will be scheduled separately so that the final locations of the STORE and LOAD will not generally be in adjacent instructions.

Note that these expressions may be as simple as semantically equivalent operations derived from a simple understanding of the target architecture or as complex as some of the esoteric operation sequences generated by a “superoptimizer” [Mas87]. The actual expression chosen from `MUTATIONS(VAL)` to re-generate `VAL` depends on the relative merits of accessing memory versus re-computing the value, which in turn depends on the functional, register, and memory bandwidth availability *at that time* in the scheduling process.

Paper Outline

Section 2 relates Mutation Scheduling to other techniques that redefine, or at least blur, boundaries that have traditionally separated code selection, register allocation, and instruction scheduling. Section 3 describes the context in which Mutation Scheduling is instantiated and describes how “values” are represented. Section 4 describes the MS technique itself: Section 4.1 describes how `MUTATIONS` sets are implemented, Section 4.2 discusses techniques for constructing the functionally equivalent expressions contained within `MUTATIONS` sets, Section 4.3 defines the core MS transformation, `MUTATE`, and, finally, Section 4.4 describes how `MUTATE` is used to perform the actual scheduling. Section 5 presents simulation results for a few target architectures that highlight different aspects of the resource trade-offs provided by MS.

2 Related Work

In recent years, a number of techniques have been developed that, to one extent or another, redefine the boundaries between the conventional separation of code selection, register allocation, and instruction scheduling concerns. For example, instruction scheduling and register allocation are integrated in [NE90, NPW91, ME92, NN92b, BGS93, BGS94]. Techniques like [NE90, NPW91, ME92, NN92b] start with an initial register allocation and then during scheduling allocate unused registers using dynamic renaming [CF87] to remove false dependencies.⁵ Unlike these techniques, which do not release allocated registers for use in other computations (e.g. by introducing spill code), the technique presented in [BGS93, BGS94] does do full register allocation and instruction scheduling “on the fly”, including register spilling when appropriate.

The *re-materialization* technique presented in [BCT92] is a register allocation technique that, unlike the above methods, partially integrates the code selection pertaining to re-generating needed values. In this technique, a conventional register allocation phase is performed, but when a live value needs to be removed from the register file, the operation that originally computed the value may be used to re-compute it in place of using spill code so long as doing so is worthwhile by some heuristics measure and semantics preserving. Some traditional compiler optimizations, like Strength Reduction and Common Subexpression Elimination (CSE) [ASU86] exhibit similar trade-offs between code selection and register allocation, but traditional compilers eliminate the *trade-off* (e.g. by performing the transformation prior to or after register allocation).

---

⁵Some of these techniques also try to eliminate spill code that becomes unnecessary as a result of scheduling.

Finally, Incremental Tree-Height Reduction (ITHR) partially integrates code selection into instruction scheduling. THR[KMC72] is a well-known technique for changing the structure of expressions by exploiting the associative and distributive properties of most arithmetic and logical operators. *Incremental* THR[NP91] was used to change (on the fly) dependencies encountered during instruction-level scheduling when doing so would increase the degree of parallelism.

Each of the abovementioned techniques has successfully out-performed techniques that rely on the conventional separation of concerns when compiling for fine-grain parallel architectures, and collectively, these successes provide much of the motivation for Mutation Scheduling (MS). MS differs most notably from the previous techniques in that it unifies all three aspects, code selection, register allocation, and instruction scheduling, into a single framework in which trade-offs among all three can be made, but MS does share some similarities with each technique:

- Like the techniques used in [BGS93, BGS94] MS attempts to do full register allocation in conjunction with instruction scheduling, but unlike these techniques, MS follows the same paradigm as used in [NE90, NPW91, ME92, NN92b], of starting with an initial register allocation and then modifying it during scheduling in response to changing resource availability.
- Like re-materialization, MS allows code other than a spill/reload sequence to re-generate a value previously removed from the register file, but MS allows the use of any functionally equivalent computation, the choice of which (including spill code) depends on the relative merits of each with respect to current resource availability at the current point during scheduling.
- Like the incremental THR technique, MS allows the code used to compute expressions to change during scheduling in response to changing resource dependencies, but does so with greater generality as any functionally equivalent expression may be substituted,⁶ not just those derivable using THR.

Of course, like the previous techniques, MS does by necessity depend on heuristic guidance since the problems it deals with are each NP-hard. However, MS has the advantage of allowing a single uniform heuristic to provide trade-offs among code selection, register allocation, and instruction scheduling within a unified framework. Within this framework, the heuristic aspects are encapsulated away from the actual code transformations so all three problems can easily be tuned by adjusting the heuristics, and without modifying the code transformation algorithms themselves.

3 Value-oriented TiPS

MS schedules operations using a system of parallelizing program transformations called Trailblazing Percolation Scheduling (TiPS)[NN92b]. TiPS is a hierarchical extension of Percolation Scheduling (PS)[Nic85] that preserves

---

⁶Including those derivable using CSE, re-materialization, and strength-reduction, as well as more complex sequences such as those that may be derived by a programmer with expert knowledge of the target architecture or a super-optimizer[Mas87].

the completeness of PS while enabling non-incremental transformations on a type of hierarchical control flow graph called the Hierarchical Task Graph (HTG)[Gir91, GP92]. HTG's were originally proposed by Girkar and Polychronopoulos for use at the coarse-grain level, but we have adapted them for use at the instruction level. In our adaptation of HTG's, *simple* nodes contain sets of machine operations that can be executed in parallel. The maximum number of each kind of operation allowed in each simple node depends on the specific target architecture. For example, a superscalar with one integer ALU, one floating-point ALU, and a branch unit, would allow at most one of each type of operation (integer, floating-point, and branch) in each simple node. Other types of nodes in the HTG, called *compound* nodes and *loop* nodes, provide the "hierarchy" in the HTG's by containing sub-HTG's composed of other HTG nodes, including simple, compound, and loop nodes.

To facilitate the "value-oriented" view used by MS, we augment the HTG structure with a slightly modified version of the Static Single Assignment (SSA) form of [CFR⁺91]. In the pure SSA form of a program, each use is reached by exactly one definition. When definitions from multiple control paths are needed by a use, a special ϕ -function is inserted at the confluence of the control paths that merges the multiple values into a single definition. In SSA form, each variable name corresponds to a value that is defined once and used at least once. At any given time during execution, the value defined by a ϕ -function is just the ϕ -function's use which is reached along the current control path. The unique name for each definition in SSA form makes it natural to view each variable as a "value" produced and consumed, in the dataflow sense, by its definition and uses. For this reason, SSA variables are usually referred to as "values" in this paper.

For simplicity of exposition, we assume that LOAD's and STORE's of elements of arrays and structures are represented using the Access, Update, and HiddenUpdate operations defined in [CFR⁺91] which, in terms of dataflow, treat each element LOAD/STORE as a scalar read/write of the entire array or structure.⁷ Obviously, reading and writing entire arrays or structures each time an element is accessed is wildly impractical for any existing system, therefore these semantics are only used in terms of maintaining the SSA form — the execution semantics of Access, Update, and HiddenUpdate operations are exactly the same as those of the element LOAD or STORE that they represent. Unfortunately, this implies that the true dependencies represented by the SSA def-use relationship is not sufficient to completely specify interdependencies among memory operations. Fully determining interdependencies among memory operations requires that anti- and output- dependencies, which do not exist among scalar reads and writes in SSA form, must also be tested for. The impact this has on MS is that potentially ambiguous memory writes are only scheduled earlier than a preceding memory read or write if disambiguation techniques can guarantee that the accesses are independent. Similarly, we do not speculatively schedule potentially ambiguous memory writes unless we can guarantee that doing so would not over-write any live information. The issues involved in dealing with potentially ambiguous memory accesses are pervasive and often difficult but do not impact MS to any greater extent than any other scheduling or parallelization techniques. Note that provably unambiguous memory accesses, such as the spill/reload sequences that may be generated by

⁷The actual method used in our compiler is very similar to this, but allows for less conservative handling of ambiguous references and incremental updates of SSA form. While the issues involved in handling ambiguous references are somewhat interesting in the context of SSA form, they are not especially relevant to MS itself and will not be discussed further in this paper.

MS during scheduling, can be treated exactly as any other true scalar operation by treating the memory location being accessed as a virtual register (i.e. true scalar variable).

The parallelizing transformations provided by SSA enhanced TiPS are essentially the same as those of normal TiPS. The only significant difference is in maintaining the SSA form as scheduling progresses. In order to expose more parallelism during scheduling, we often allow operations below the join-point of a conditional to “move” into the conditional by duplicating them into its “true” and “false” branches. If such an operation, OP, depends on a ϕ -function at the join-point of a conditional, then OP is control dependent on the conditional and may use and define different values depending on which branch is taken through the conditional. If OP is copied to OP' along one branch, say B, of the conditional, then each use in OP' that depends on the ϕ -function will be replaced by the ϕ -function's use that is reached through B. Similarly, if OP defines the variable X then the copy of OP on each branch will define a new variable, say X' and X'' respectively, and the ϕ -function $X \leftarrow \phi(X', X'')$ will replace OP at the join-point. Note that if OP does not depend on any ϕ -function at the join-point, then OP would not be control dependent on the conditional and identical copies of OP could be duplicated onto each control path, without requiring a new ϕ -function — this is a slight structural relaxation of true SSA form which requires a unique destination name for each *textual* occurrence of an operation, but is fundamentally consistent with the intended meaning of SSA form since each *value* is still given a unique name. Note that this relaxation also allows multiple textual occurrences of the same value to occur when “re-generating” values that have been released from the register file.

4 Mutation Scheduling

This section describes Mutation Scheduling. MUTATIONS sets are defined in Section 4.1. Techniques for finding and constructing multiple functionally equivalent “mutations” for values are discussed in Section 4.2. Section 4.3 describes the MUTATE transformation that provides the mechanism for using the MUTATIONS sets for integrating both code selection and register allocation into the scheduling process as detailed in Section 4.4.

4.1 Mutation Sets

Conceptually, we view each MUTATIONS set as a set of arbitrarily long expressions, each of which computes the same value. However, since the intermediate values defined within any expression may themselves have multiple expressions (i.e. mutations), we represent expressions as follows. Given an expressions, E, that computes VAL, the *tree form* of E consists of *leaf* nodes and *interior* nodes. Each leaf corresponds to either a constant or a value produced for E by some other expression. Each interior node corresponds to an operation in E that uses the values defined by its “children” in the tree and defines a value for use by its “parent” in the tree. Edges in the tree join each operand used by an operation (i.e. interior node) to its definition: either a leaf or the destination operand of some other operation. The root of the tree corresponds to the operation that defines VAL. This tree form expression for VAL is represented in our system using MUTATIONS sets by storing the

operation, OP, associated with the root of the tree in MUTATIONS(VAL) and creating new MUTATIONS sets for each (sub)expression rooted at a non-leaf child associated with a use in OP. We assume that each expression is already in SSA form with respect to the program (even though the expression itself may not be present in the program) so any new MUTATIONS sets created will be for new values.

Note that in the set theoretic sense, MUTATIONS(VAL) only contains *operations*; however, in the remainder of this paper, we will say an *expression*, E , is in MUTATIONS(VAL) if E can be re-constructed from the MUTATIONS sets by reversing the above process, starting with an operation from MUTATIONS(VAL) as the root of the tree for E . This representation of expressions is similar to value numbering[ASU86] except that instead of representing multiple identical subexpressions as a single subexpression, referenced multiple times within a directed acyclic graph, we represent multiple identical equivalence classes of expressions (i.e. MUTATIONS sets) as a single equivalence class referenced multiple times by its value name.

4.2 Finding Expressions

Any technique may be used for generating the initial set of expressions stored in the MUTATIONS sets as long as the abovementioned guidelines are followed. One simple, but useful technique is to initialize the MUTATIONS set for each value, VAL, defined by the operation, OP, to $MUTATIONS(VAL) = \{OP\}$, and then take the transitive closure of each MUTATIONS set with respect to a (constructive) functional equivalence relation, $\cong$, defined as follows. Given expressions A and B in tree form, $A \cong B$ if and only if all of the following are true:

1. The leaves of A and B reference the same set of variables (i.e. A and B have the same "inputs").
2. The operations associated with the roots of A and B define the same variable (i.e. they have the same "outputs").
3. For all valid assignments to the leaf variables of A and B, the values produced by A and B are identical, and this equivalence can be determined by the compiler (thus $\cong$ is a "constructive" relation).

Note that this definition says nothing about the internal structure of either expression which may be arbitrarily complex or simple. For instance, THR can often yield a large number of functionally equivalent expressions,⁸ each of which may have different numbers and kinds of operations in addition to a different structure.

The rules used in step 3 to determine functional equivalence will, by necessity, be incomplete due to the undecidability of the "program equivalence" problem; however, for any given target architecture, simple but important classes of functional equivalence can usually be determined. For instance, the following three types of functional equivalence generally lend themselves to constructive characterization for most architectures and application domains:

Synonyms When restricted to specific functions, many operations have "synonymous" meanings. For instance, " $X \leftarrow Y$ ", can be performed on most architectures, in a number of ways: (ASSIGN X Y), (ADD X Y 0),

⁸Although, due to the finite precision of computer arithmetic, the actual values produced at run time may differ somewhat for some THR-equivalent expressions. Whether or not these differences are significant is application dependent.

(MUL X Y 1), (OR X Y 0), etc. Each of which may require different resources, and may or may not have different latencies depending on the particular architecture.

Composition/decomposition Many architectures offer specialized “compound” operations that perform the same function as multiple sequences of other operations. It is often possible to change the resource requirements and latency of a given computation by “composing” such a sequence of operations into an equivalent compound operation, or by “decomposing” a compound operation into a sequence of operations. For instance, most processors designed for use on Digital Signal Processing applications offer a single Multiply And Accumulate (MAC) operation that performs the computation $A \leftarrow B + C * D$, which could also be “decomposed” into the two operation sequence $t \leftarrow C * D, A \leftarrow B + t$. Executing the MAC operation takes less time than the decomposed pair of operations; however, the MAC operation typically requires MULTIPLY and ADD functional units to be used “back to back”, which may not be possible at a particular point in the program if some other operation has already been scheduled onto one of those units. In this case, if the MULTIPLY and ADD units were separately available then the decomposed sequence of operations might produce the same result sooner.

In addition, some architectures, such as the Intel i860, provide a more general mechanism that allows fairly arbitrary “compositions” of operations executed on some functional units by allowing an operand address to specify that the output of one functional unit be fed directly into the input of the same or another unit, thus bypassing register access stages of the execution pipe. This type of explicit register-file bypassing feature is complicated by the fact that the result from a functional unit is usually only available for bypassing during a specific “hot-spot” cycle in which the recipient must access the value. Fortunately, Mutation Scheduling provides a trivial mechanism for performing “hot-spot” scheduling since it is possible to keep both the hot-spot and register-to-register mutations of an expression for a given value within the MUTATIONS set of the value. Then, at any point during scheduling, the mutation of the value that best utilizes the available resources can be selected.

Even for conventional RISC processors with no such “specialized” operations, composition and decomposition can still sometimes be used. For instance, the expression $X \leftarrow Y * 3$ could be implemented using the single operation (MUL X Y 3), but when the multiply unit is busy computing other values, it might be worthwhile to decompose this operation into the pair of operations, (ADD t Y Y), (ADD X t Y).

Tree Height Reduction and Constant Folding Tree Height Reduction (THR) is a well-known technique for changing the structure of an expression in tree form by exploiting the associative and distributive properties of most arithmetic and logical operations [KMC72]. By “flattening out” tall thin expression trees into shorter, wider ones, THR increases the degree of parallelism in the expression, and thereby changes its resource requirements.

Similarly, when the expression contains constant terms, it is often possible to group the constant terms together into the same operation using THR. This then allows the operations involving only constant

terms to be performed statically at compile time and replaced by the new resulting constant. This process is typically referred to as Constant Folding. Of course, neither THR nor Constant Folding should be performed if it would result in numerical instability for a given application.

When the above types of functional equivalence combine as mutations during scheduling they may also have the cumulative effect of performing other conventional optimizations. For instance, constant folding, composition/decomposition, and synonym mutations sometimes combine to yield an incremental form of strength reduction.

Other types of functional equivalence might also be employed, including some of the more esoteric “tricks” that might be provided by an experienced assembly language programmer or a super-optimizer[Mas87]. However, even the above “simple” types of functional equivalences are sufficient to yield a large number of functionally equivalent expressions. For instance, even an expression as simple (and common) as the address computation for the array reference, $M[i][j]$, exhibits all three forms of functional equivalence. If C_1 is the row size of M and C_2 is the element size of M , then the address of $M[i][j]$ is $\text{ADDR} \leftarrow (i * C_1 + j) * C_2$, which, after Synonym, Composition/Decomposition, and THR closure, is represented using MUTATIONS sets as:

$$\begin{aligned} \text{MUTATIONS}(\text{ADDR}) = \{ & (\text{MUL ADDR } t_1 \text{ } c_2), (\text{LSHIFT ADDR } t_1 \text{ } \log C_2), \\ & (\text{ADD ADDR } t_3 \text{ } t_4) \} \end{aligned} \quad (1)$$

$$\text{MUTATIONS}(t_1) = \{ (\text{ADD } t_1 \text{ } t_2 \text{ } j), (\text{MAD } t_1 \text{ } j \text{ } i \text{ } C_1) \} \quad (2)$$

$$\text{MUTATIONS}(t_2) = \{ (\text{MUL } t_2 \text{ } i \text{ } C_1) \} \quad (3)$$

$$\text{MUTATIONS}(t_3) = \{ (\text{MUL } t_3 \text{ } i \text{ } C_1 * C_2) \} \quad (4)$$

$$\text{MUTATIONS}(t_4) = \{ (\text{MUL } t_4 \text{ } j \text{ } C_2), (\text{LSHIFT } t_4 \text{ } j \text{ } \log C_2) \} \quad (5)$$

The above assumes that C_2 is a power of two, which is typical for arrays containing scalar values. Note that the MAD operation ($\text{MAD } A \text{ } B \text{ } C \text{ } D$) has the meaning $A \leftarrow B + C * D$. Ignoring resource constraints, the expression “($\text{MAD } t_1 \text{ } j \text{ } i \text{ } C_1$), ($\text{LSHIFT ADDR } t_1 \text{ } \log C_2$)” would generally be selected during the initial code selection as the best expression for ADDR; however, at any given point during scheduling, the computation that would actually produce ADDR sooner may very well be one of the other mutations if the resources required for one or both of these operations are unavailable. An example of how this can happen is found in Section 4.3.

4.3 The Mutate Transformation

The core transformation of MS is the MUTATE routine shown in Figure 1. MUTATE is responsible for selecting and scheduling the “best” computation of “VAL”, if any, that can be scheduled on the path of instructions described by the Path-Template, P . A Path-Template is a list of 3-tuples of the form (NODE, OPS, RESOURCE-CONSTRAINTS). Each such tuple contains an instruction, NODE, a set of operations, OPS, that would need to be added to NODE in order to compute VAL, and a list of resource constraints, RESOURCE-CONSTRAINTS,

```

function MUTATE(VAL, P)
  — try shorter mutations first
  for i  $\leftarrow$  1 ...  $|P|$  do
    if REALIZABLE-MUTATION(VAL, (P1, ..., Pi)) then
      REALIZE-MUTATION(VAL, (P1, ..., Pi))
      return true
    end if
  end do
  return false
end function

function REALIZABLE-MUTATION(VAL, P)
  if VAL in LIVE(NODE(P1)) then
    return true
  end if
  while OP  $\leftarrow$  CHOOSE-OP(MUTATIONS(VAL)) do
    if OPS(P1)  $\cup$  { OP } satisfies RESOURCE-CONSTRAINTS(P1)
      and (Reads(OP)  $\subseteq$  LIVE(NODE(P1)) or
         $|P| > 1$  and ( $\forall U \in \text{Reads}(\text{OP}), \text{REALIZABLE-MUTATION}(U, (P_2, \dots, P_{|P|}))$ ))
      then
        OPS(P1) = OPS(P1)  $\cup$  { OP }
        return true
      end if
    end do
    — VAL is not realizable, so forget any subexpressions
    for i  $\leftarrow$  2 ...  $|P|$  do
      OPS(Pi) =  $\emptyset$ 
    end do
    return false
  end function

procedure REALIZE-MUTATION(VAL, P)
  for i  $\leftarrow$   $|P|$  ... 1 do
    if |Preds(NODE(Pi))| > 1 then
      ISOLATE(NODE(Pi))
    end if
    Insert(OPS(Pi), NODE(Pi))
    REMOVE-REDUNDANCIES(NODE(Pi))
    UPDATE-LIVE-INFO(NODE(Pi))
  end do
end procedure

```

Figure 1: The MUTATE Transformation

that would be satisfied if all of the operations in OPS were added to node. For a given node, N, functional resource constraints specify how many new operations may be scheduled for each functional unit at N, and register constraints generally specify how many new *register-values* may become live at N. A *register-value* is a value defined by an operation that writes to a register, as opposed to a *memory-value* which is defined by an operation (e.g. STORE) that writes to memory. We say that the value VAL is *allocated* a register at N if, and only if, VAL is a live register-value at N. When the target architecture has hot-spots, we treat each hot-spot as a register resource (e.g. latch register) that exists for only one cycle after it is defined (as opposed to normal register resources which always exist). This prevents MUTATE from scheduling the definition of the hot-spot more than one cycle prior to the use of the hot-spot value. Scheduling the definition earlier than this would cause the register holding the hot-spot value to cease to exist at its use, thereby causing the number of live hot-spot values to exceed the number of available hot-spot registers at the use of the hot-spot which is not allowed by the MUTATE transformation.

The first tuple on the Path-Template list corresponds to the node, N, in which the actual definition of VAL is intended to take place. Any subsequent tuples represent a path leading to N along which intermediate values needed for computing VAL may be "realized". We say a value VAL is *realizable with respect to the Path-Template, P*, if VAL is already live at Node(P_1), or some operation OP from MUTATIONS(VAL) can be scheduled at Node(P_1) such that the resource constraints, RESOURCE-CONSTRAINTS(P_1), are satisfied and each of the values read by OP are realizable with respect to the rest of the Path-Template after P_1 , i.e. ($P_2, \dots, P_{|P|}$). The REALIZABLE-MUTATION routine shown in Figure 1 is a constructive version of this condition. When called from MUTATE, the OPS sets of each tuple in the Path-Template are empty. As REALIZABLE-MUTATION searches for a realizable mutation, the OPS sets are filled in with the operations that would be needed to compute, or *realize*, VAL. If REALIZABLE-MUTATION succeeds, then REALIZE-MUTATION inserts the operations contained in the OPS set of each tuple on P into the corresponding NODE, removes any redundancies⁹ thus created, and updates the live information at the nodes affected. Any affected nodes on more than one control path are duplicated so as to isolate the effect of the new expression to the path specified in P.

Notice that MUTATE always prefers shorter expressions over longer ones, but the maximum length of expressions, the resources they consume, and the order in which mutations are tried all represent heuristics that are completely separate from the MUTATE transformation itself, thus making MUTATE a general transformation that could be used with most, if not all, approaches to instruction scheduling. The specific scheduling approach, in conjunction with the desired application domain and cost vs. performance trade-offs would dictate the actual choices for the MUTATE heuristics.

Figure 2 shows an application of the MUTATE transformation to realize a mutation of the value, ADDR, whose MUTATIONS set was defined on page 9. Recall that the best mutation for ADDR in the absence of

⁹If the target architecture supports conditional write-back as in [Ebc88], then it is possible that two operations from the same instruction produce the same value such that the write-back of one operation is predicated on some condition, and the write-back of the other on the negation of the condition. In this case, both operations can be replaced by a single operation whose write-back is not predicated on either the condition or its negation.

MUTATE(ADDR, (< N, $\emptyset$, "1 add unit and 2 registers available" >, < M, $\emptyset$, "1 add, 1 mul and 1 shift unit available" >))

Figure 2: MUTATE example

resource constraints was "(MAD $t_1 j i C_1$), (LSHIFT ADDR $t_1 \log C_2$)". However, REALIZABLE-MUTATION can not return this mutation since the shift unit is already assigned an operation at N. The expression rooted at "(MUL ADDR $t_1 c_2$)" might also be considered, but this expression is also not realizable at N since the multiply unit is also busy. This leaves the final operation in MUTATIONS(ADDR), "(ADD ADDR $t_3 t_4$)". This operation does satisfy RESOURCE-CONSTRAINTS(N) but is not yet ready to execute at N since its uses, t_3 and t_4 , are not live at N. In this case, since the Path-Template also includes a 3-tuple for M, REALIZABLE-MUTATION will recursively try to realize t_3 and t_4 at M. The value t_3 is realizable at M since a functional unit is available at M for "(MUL $t_3 i C_1 * C_2$)" and a register is available to hold t_3 on entry to N. For t_4 , there are two candidate mutations, "(MUL $t_4 j C_2$)" and "(LSHIFT $t_4 j \log C_2$)". Since the multiply unit has already been (virtually) assigned to the computation of t_3 , the first operation is not realizable at M. The shift operation is realizable at M, since the shift unit is available at M and an available register remains at N to hold t_4 . The right side of Figure 2 shows the mutation of ADDR that was realized at N by MUTATE.

4.4 Scheduling

This section details how the MUTATE transformation is integrated into an existing Global Resource-constrained Percolation (GRiP)[NN92a] scheduler to yield a Mutation Scheduling system. GRiP scheduling requires that

```

procedure MOVE-VAL(VAL, OP, FROM, TO)
  if VAL not in LIVE(Node(TO)) then
    if Writes(TO)  $\cap$  Reads(OP)  $\neq \emptyset$ 
      or not Enough-Resources(OP, TO) then
        if not MUTATE(VAL, GET-CONT-PATH(TO)) then
          return
        end if
      else
        Insert-op(OP, TO)
      end if
    end if
    if |Preds(FROM)| > 1 then
      ISOLATE(FROM)
    end if
    DELETE-OP(OP)
    UPDATE-LIVE-INFO(FROM)
    if TOO-MANY-LIVE-REGS(LIVE(FROM)) then
      DECREASE-REGISTER-PRESSURE(FROM)
    end if
  end procedure
procedure DECREASE-REGISTER-PRESSURE(FROM)
  while VAL  $\leftarrow$  CHOOSE-REGISTER-VALUE(LIVE(FROM)) do
    if VAL not in Reads(FROM) and MUTATE(VAL, GET-REGEN-PATH(FROM)) then
      return
    end if
  end do
end procedure

```

Figure 3: The MOVE-VAL transformation

satisfiable resource constraints be an invariant between code motion transformations.¹⁰ Using GRiP, operations are progressively scheduled earlier using PS or TiPS transformations, in ranked order, until blocked by resource constraints (i.e. resource dependencies), true data dependencies, or false data dependencies when no free registers are available for performing dynamic renaming. In Mutation Scheduling, whenever one of these dependencies is encountered during scheduling, MUTATE is used in an attempt to remove the dependence. If an operation that defines *VAL* is prevented from being scheduled earlier by a true data dependence or functional resource dependence, then we try to find a new mutation of *VAL* that can be scheduled earlier. If code motion is blocked by a lack of available registers, then MUTATE may be used to free one or more registers by re-generating the values stored in these registers at some later time.

The MOVE-VAL routine shown in Figure 3 is an incremental transformation¹¹ responsible for trying to schedule

¹⁰As opposed to techniques such as [Pot91] that ignore resource-constraints during scheduling and [GS90, BGS93, BGS94] that satisfy only resource estimates.

¹¹For the sake of notational convenience, we will describe MS in the context of modifying the incremental PS transformation,

the definition of the value VAL, which is currently defined by the operation OP in the instruction FROM, one instruction earlier at TO.¹² Because of the “relaxed” SSA form, which allows multiple definitions of the same variable as long as the value defined by each is the same, it is possible that some other definition of VAL already reaches TO (i.e. $VAL \in \text{LIVE}(TO)$). This can happen if a definition of VAL has already been scheduled separately along some other control path through TO, or if VAL was previously defined along some control path leading to FROM, but had to be kicked out of the register file by generating code (including the definition of VAL in FROM) to re-generate VAL. In either case, the presence of VAL in $\text{LIVE}(TO)$ when a definition of VAL also exists in FROM indicates that the conditions that originally required that the (re)definition of VAL currently scheduled at FROM have changed so that OP can simply be deleted, thus causing any definition of VAL that reaches TO to also reach the uses that were reached by OP. This facility for “extending” the live range of previous definitions of the same value provides a trivial but useful mechanism for both performing the PS UNIFY transformation,¹³ and for “undoing” bad register allocation decisions by eliminating any regeneration code that becomes unnecessary due to changing register pressure during scheduling.

If VAL is not live at TO, then MOVE-VAL begins by testing for either a functional resource or data dependence on TO. If neither dependence exists then the current definition of VAL can simply be scheduled at TO. If a dependence does exist, then MUTATE tries to realize a mutation of VAL at TO. The GET-CONT-PATH function returns a Path-Template for a path of nodes ending in TO along which the computation of VAL may take place.¹⁴ The resource constraints specified within the Path-Template must never exceed the maximum resources available along the specified path of instructions, but may be much less depending on the relative importance of the value being mutated and the resources that would be freed (from the current mutation of the value) if a new mutation were to be scheduled earlier.

If MUTATE fails to realize VAL at TO, then VAL is not currently “movable” and is suspended until the condition that blocked its movement changes. If a definition of VAL is successfully scheduled at TO (possibly by MUTATE), then the effect of the movement must be “isolated” to the current path along which VAL is being scheduled. If FROM has multiple predecessors then ISOLATE inserts a copy of FROM on all other paths before DELETE-OP (which is called anyway) deletes OP from the copy of FROM that now has no predecessors other than TO. DELETE-OP will also recursively delete any definitions that were only used by OP. If OP was the only operation in FROM then FROM no longer exists and we are done. Otherwise, live information is updated to reflect the fact that VAL has become, and other values have possibly ceased to be, live at FROM. If it turns out that the number of live register-values now exceeds the actual number of registers, as might happen, for instance,

---

MOVE-OP rather than its non-incremental counter-part (TRAILBLAZE) used in TIPS (the MS specific issues are identical for both).

¹²Note that VAL must be a register-value or a memory-value — hot-spot values are scheduled solely as a byproduct of the MUTATE transformation.

¹³UNIFY merges multiple definitions that have reached the same point after being scheduled separately along different control paths.

¹⁴Recall that the length of this path represents the *maximum* length of the expression — the actual mutation is often just one operation. Note that a maximum path length of two is sufficient to exploit most Synonym, Composition/Decomposition, and THR mutations.

if VAL is a register-value and no other register-values ceased to be live, then DECREASE-REGISTER-PRESSURE is called to restore the satisfiability of register constraints.¹⁵

DECREASE-REGISTER-PRESSURE applies some selection criteria (e.g. “furthest next use”) to choose register-values from LIVE(FROM) in increasing order of importance. The first such register-value, say X, that is both unused in, and realizable at, FROM will be scheduled at FROM to kill X and thereby “release” the register allocated to X (i.e. X will no longer be live at the entry to FROM). As long as the selection criteria ensures that all register-values are candidates for selection, then DECREASE-REGISTER-PRESSURE will always succeed in restoring correct register utilization since VAL, which must be realizable at FROM, may be selected as the register-value to release. In any case, we delete any previous definition of X that is killed by the new definition of X just scheduled at FROM. Repeated calls to DECREASE-REGISTER-PRESSURE (when the register file is saturated) have the cumulative effect of deferring the definitions of values to the latest point at which they would be useful.

5 Results

Mutation Scheduling “adapts” code to the resources of the target architecture by making trade-offs, on the fly, between functional, register, and memory bandwidth resources in response to changing constraints and availability during scheduling. In this section, we present results from experiments for a few different target architectures that highlight three important aspects of these trade-offs. In the first, we focus on register versus functional resource and memory bandwidth trade-offs; in the second, we focus on parallelism versus register and functional resource trade-offs, and in the third, we focus on trade-offs among heterogeneous and specialized functional units.

Each experiment compares the speedup¹⁶ obtained using a “Mutating” GRiP compiler (as outlined in section 4) for a set of benchmarks against those obtained using a non-mutating GRiP compiler. The presence or absence of Mutation Scheduling transformations is the only difference between the two compilers — each uses the same list scheduling heuristics to rank the order of operation importance and both pipeline loops using the “Resource-Directed Loop Pipelining (RDLP)” technique described in [NN94]. RDLP works by unrolling and shifting¹⁷ loops during scheduling to expose more operations to parallelize until resources are fully utilized and/or cost vs. performance constraints are satisfied.

For all three experiments, the same exact Mutating and Non-mutating compilers were used on the same set of benchmarks, so the experiments differ only in the characteristics of the target architectures specified for each. For the first two experiments (Sections 5.1 and 5.2) we assume idealized VLIW architectures with homogeneous, unicycle functional units. These simple though perhaps unrealistic architectures provide a useful framework

---

¹⁵ A more efficient, but less elegant way of handling the register resource would be to determine, prior to scheduling VAL at TO, whether or not doing so would over-utilize the registers and then choose between aborting the move or adding code to free a register for VAL, according to the relative importance of VAL to those values already allocated registers at FROM.

¹⁶ i.e. the ratio of sequential to parallel cycles observed during simulation.

¹⁷ Loop shifting refers to “unwinding” a loop so that its head becomes a true successor of each of its predecessors, thus exposing new operations from subsequent iterations at the end of the loop for scheduling in parallel with existing operations.

for illustrating the ability of Mutation Scheduling to make general trade-offs between resources and parallelism and explicitly factor out its ability to exploit the unusual and specialized features that exist for many realistic architectures, which will be discussed separately in Section 5.3. In the third experiment we use a realistic but hypothetical VLIW architecture that combines some of the functional unit characteristics of the Motorola 88110 Superscalar with explicit pipeline and datapath control characteristics similar to those found in architectures like the Intel i860. We use this model to highlight the ability of Mutation Scheduling to exploit and make trade-offs among heterogeneous functional units and specialized architectural features.

Amdahl's law states that when optimizing the bottleneck in some task, at some point, another part of the task will become the bottleneck. This means that even a transformation that optimally schedules one part of a program will have a limited (though often substantial) effect on the overall performance of the program. A corollary of this is that when developing and analyzing an optimization for one part of the task, it is essential to factor out, as much as possible, the effect of the unoptimized portion of the task in order to get the clearest possible performance feedback about the effectiveness of the optimization in question. For this reason we have chosen to run each of our experiments on the Livermore suite. In addition to factoring out distorting (albeit interesting) characteristics, such as cache performance and i/o, these benchmarks are small and well known so the results can easily be understood and interpreted. Below we present each experiment in detail.

5.1 Registers vs. Functional Units and Memory Bandwidth

In the first experiment we focus on the ability of Mutation Scheduling to decrease register pressure when necessary by exploiting unused functional resources and memory bandwidth to recompute or spill/reload values that are removed from the register file. To highlight this ability we compiled each benchmark for a VLIW with eight homogeneous functional units and a "minimal" number of registers. For each benchmark, we define the "minimal" number of live registers allowed during parallelization (i.e. the assumed register file size of the target architecture) to be equal to the maximum number of (virtual) registers live at any point in the initial sequential schedule produced by our GNU C front-end with register allocation disabled. This is the minimum number of registers that the target architecture would need to execute the sequential schedule without spilling registers. By allowing each sequential schedule to have as many registers as it needs, we enable the frontend to produce the best schedule that it can with respect to conventional optimizations, such as strength reduction and common sub-expression elimination, and we factor out sequential register allocation constraints on the resulting parallel code. This in turn helps us ensure that speedups represent the ability of each compiler to parallelize good initial schedules rather than to merely remove deficiencies from poor ones.

Table 1 shows the speed-ups obtained when compiling each benchmark using a Mutating version of the GRiP compiler versus a Non-mutating version. Notice that Mutation never does worse than, and in all cases but one, does strictly better than the Non-mutating version. In fact, for 6 of the 14 benchmarks, the Mutation Scheduling version provides between a 2 and 3 fold improvement over the Non-mutating version. In both compilers, renaming to remove false dependencies among register-to-register operations is accomplished by SSA form, so each version

bench	RF Size	Speed-up	
		Mutating	Non-mutating
LL1	10	4.94	2.49
LL2	20	5.32	3.39
LL3	6	3.98	3.98
LL4	9	5.30	3.59
LL5	6	5.47	2.75
LL6	14	6.19	2.08
LL7	15	5.04	2.38
LL8	37	5.46	3.10
LL9	14	5.20	2.64
LL10	5	3.27	2.98
LL11	6	8.81	2.99
LL12	6	4.48	2.99
LL13	13	4.54	2.29
LL14	20	3.94	2.47
Avg	13	5.14	2.87

Table 1: Trading functional units and memory bandwidth for registers

is capable of performing the same sort of register re-allocation as is used in [NE90, NPW91, ME92, NN92b] by allowing code motion only if the number of live register values after each completed transformation does not exceed the register file size.

The 100-200% improvements often provided by Mutation Scheduling come in part from its ability to *release* registers for use by more critical operations by exploiting available functional and memory bandwidth resources to recompute or spill/reload values when needed, and also comes in part from its ability to select code that decreases register requirements by exploiting opportunities to express computations in terms of values already in the register file, in lieu of introducing new intermediate values. As an example of utilizing pre-existing values, consider induction variable (iv) computations within loops. During loop pipelining it is common for versions of iv's from multiple iterations to become simultaneously live (and therefore consuming, at least, register resources). However, in many cases, iv's are used only by operations that can participate in Constant Folding transformations, such as "base + offset" addressing in LOAD/STORE operations, equality and inequality tests against constants for loop control, and, of course, the iv itself. In this case, if the MUTATIONS sets are closed under Constant Folding as discussed in Section 4.2, it is often possible for MUTATE to consistently express mutations in terms of live iv's from the earliest possible iteration. This often has the effect of killing all but the earliest version of many iv's, thus freeing registers and functional units that had, or would otherwise have, been allocated for holding and computing these values.

bench	Speed-up	
	Mutating	Non-mutating
LL1	15.78	14.35
LL2	11.48	11.39
LL3	14.01	7.89
LL4	14.16	10.05
LL5	5.47	5.47
LL6	10.99	10.14
LL7	14.76	14.76
LL8	11.50	11.50
LL9	14.31	14.12
LL10	10.61	14.91
LL11	16.08	8.81
LL12	17.55	8.90
LL13	14.31	14.31
LL14	5.20	5.20
Avg	12.59	10.84

Table 2: Trading resources for parallelism

5.2 Parallelism vs. Register and Functional Resources

Above, we showed how MS can improve performance when the register file is heavily utilized by trading registers for available functional and memory bandwidth resources. In this experiment we show that when the architecture is robust, MS can also improve performance by increasing parallelism at the expense of increased resource consumption. For this experiment we assume a VLIW target architecture with 16 homogeneous functional units and 64 registers.

Table 2 shows the speed-ups obtained when compiling with the Mutating version of GRiP against those obtained using the Non-mutating version. Both compilers produce good speed-ups. In one case, the Non-mutating version out-performs the Mutating version, but in most of the remaining cases, the Mutating version produces significantly superior results — often improving speed-up by as much as 100%. Since the functional units are homogeneous and the register file is large enough, trade-offs between different functional units (as will be discussed in Section 5.3) and between registers and functional units (Section 5.1) do not affect the performance levels of the Mutating compiler. Thus the speed-up differences between the Mutating version of the compiler and the Non-mutating version are caused by the trade-offs between parallelism and resources provided by the Constant Folding and THR mutations defined in Section 4.2.

The Constant Folding mutations provided by MS can often increase performance by removing trivial Loop-Carried Dependencies (LCD's) on iv's. For any single threaded architecture, every loop that executes a de-

terministic and finite number of times implies at least one LCD to control the termination of the loop. Each such LCD constrains the degree of parallelism within the loop to at most the number of operations within the loop (and possibly less than this). However, when loop pipelining employs loop unrolling to expose parallelism, multiple iterations from the original loop become a single “multi-iteration” in the new unrolled loop and the successive versions of iv’s from the unrolled iterations are thus exposed to Constant Folding and possible removal as deadcode as discussed in Section 5.1. The net effect of doing this type of mutation is that many LCD’s between iterations of the original loop can be replaced by a single LCD on the “multi-iteration” of the unrolled loop so that parallelism in the unrolled loop is then constrained by the number of operations in the larger “multi-iteration”. For vectorizable loops, this process allows parallelism to be limited only by the number of iterations executed by the loop and often allows loop pipelining to continue scheduling new iterations until resources are completely utilized.

THR mutations improve performance by exploiting the associative and distributive properties of arithmetic to restructure expressions in order to increase the degree of parallelism in the expressions, sometimes at the expense of greater resource consumption (and, rarely, at the cost of numerical instability). Like Constant Folding, the major benefit of using THR mutations often occurs when used in conjunction with a loop pipelining technique that employs loop unrolling. In this case, many non-trivial LCD’s, such as those that are often caused by “summation of terms” types of computation, become long, thin expression trees after loop unrolling that can then be flattened into shorter, wider expressions (i.e. with shorter critical paths and more parallelism). Unfortunately, due to its potential for increasing resource consumption, THR mutations can also degrade program performance, as is indicated by LL10. In LL10, the THR mutations performed do increase resource consumption as well as parallelism, but since the expressions involved in the THR mutations are not part of a LCD, the Non-mutating GRiP compiler is able to achieve the same or greater increases in parallelism just by loop pipelining, without any increases in resource consumption. We believe that a more sophisticated mutation heuristic, such as “only choose THR mutations when there is no net increase in resource consumption”, would greatly decrease the possibility that MS might degrade performance, without significantly inhibiting the ability of MS to increase parallelism.

5.3 Trade-offs among Heterogeneous and Specialized Functional Units

In this experiment we focus on the ability of Mutation Scheduling to increase parallelism by making trade-offs among heterogeneous functional units and specialized architectural features. The target architecture is a hypothetical VLIW machine combining some of the functional unit characteristics of the Motorola 88110 Superscalar with some explicit instruction issue and datapath control characteristics found in a few real-world architectures like the i860. For this model we assume the functional units described in Table 3. With the exception of the branch unit, which was arbitrarily defined to take 4 cycles, each of these functional unit kinds and latencies are roughly the same as those of the Motorola 88110 Superscalar.

For this experiment we assume that there are two each of ALU, SHIFT, FALU, MUL, and MEM units, and one each of DIV and BRANCH units (i.e. just a little less functionality than a two-wide homogeneous VLIW).

Name	Latency	Description
ALU	1 cycle	integer add/sub and logical
SHIFT	1 cycle	arithmetic and logical shifts
FALU	3 cycles	floating point add/sub and logical
MUL	3 cycles	integer and floating point multiply
DIV	13 cycles	integer and floating point divide
MEM	2 cycles	cache read (cache miss stalls the processor)
	1 cycle	cache write
BRANCH	4 cycles	conditional branch

Table 3: Functional Unit Kind and Latency

We assume a single register file with 64 registers. In terms of control logic we adopt an approach similar in some respects to the i860. Each VLIW instruction specifies exactly one (possibly NOP) operation for each functional unit. Each operation issued to a functional unit can have the optional side-effect of pushing along the execution pipeline of the functional unit.¹⁸ Furthermore we assume that the data paths are set up to allow explicit register file bypassing by allowing each operand of an instruction to address either a register¹⁹ or the output of any functional unit. We assume that register fetch and write-back stages are part of the pipeline for each functional unit, and unless explicitly bypassed as mentioned above, each take one cycle (i.e. without bypassing, the latency from fetch to writeback of any operation is two greater than the execution latency of the functional unit that executes the operation).

Table 4 shows results obtained for this architecture. Both the Mutating and Non-mutating GRiP compilers perform well, but the Mutating version consistently out-performs the Non-mutating version, often by as much as 50%. Note that even though this machine model has essentially the same functional resources as a 2-wide homogeneous VLIW, the Mutating GRiP compiler, and to a lesser extent the Non-mutating GRiP compiler, often produce order of magnitude speed-ups over the sequential case. This is indicative of the ability of each compiler, but especially the Mutating version, to effectively exploit both the spatial and temporal (pipeline) parallelism of the target architecture.

One of the main reasons for the large improvements of Mutation Scheduling over the Non-mutating compiler for this target architecture is the ability of Mutation Scheduling to make better utilization of explicit register file bypassing. One interesting thing to note however is that at times explicit register file bypassing can actually degrade the overall performance of the code, even though locally it always decreases the latency of a computation.

¹⁸Even though explicit pipeline control has proven difficult, or at least inconvenient, for many compilers, Mutation Scheduling provides a fairly clean mechanism for scheduling in this environment which allows us to exploit some of the benefits of explicitly controlling the pipeline. For instance, it is possible to limit the effect of any multi-cycle operation to specific control paths so that at targets of conditional branches there are no unwanted structural or data hazards on previously issued (but incomplete) multi-cycle operations.

¹⁹In this context, we treat any immediate fields within the instruction as registers.

bench	Speed-up	
	Mutating	Non-mutating
LL1	11.92	10.70
LL2	8.99	8.93
LL3	7.19	5.76
LL4	5.06	4.22
LL5	6.69	4.61
LL6	6.61	4.98
LL7	10.06	9.97
LL8	9.01	5.92
LL9	4.33	2.85
LL10	2.84	2.53
LL11	5.54	5.04
LL12	7.95	9.24
LL13	3.46	2.75
LL14	3.74	2.53
Avg	6.67	5.72

Table 4: Trade-offs among heterogeneous and specialized functional units

To see this consider bypassing the register file for a floating point multiply/add sequence. This decreases the latency of the sequence by two, from 10 total cycles to 8, thus improving the utilization of temporal parallelism. However, loop pipelining could have scheduled the two operations (from different iterations) in parallel spatially, in which case the latency of the sequence would, in some sense, have been only 5 cycles (the total fetch to writeback latency of each operation). The Mutation Scheduling heuristics attempt to handle this trade-off by “unbypassing” operations when doing so would likely improve performance, but LL12, which is the only benchmark for which the Mutating version did not do strictly better than the Non-mutating version, is one example where the Mutation heuristics over-zealously exploited the bypassing feature and thus constrained the overall performance of the loop. It should be emphasized that we currently use very simple Mutation heuristics that could easily be improved upon; however, an unfortunate reality of dealing with NP-hard problems, such as resource constrained scheduling, is that no matter how good the heuristics get there will always be some cases for which they perform sub-optimally.

References

- [ASU86] A. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1986.
- [BCT92] P. Briggs, K. Cooper, and L. Torczon. Rematerialization. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, 1992.
- [BGS93] D. Berson, R. Gupta, and M.L. Soffa. Ursa: A unified resource allocator for registers and functional units in vliw architectures. In *Proceedings of the IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, Orlando, FL, January 1993.
- [BGS94] D. Berson, R. Gupta, and M.L. Soffa. Resource spackling: A framework for integrating register allocation in local and global schedulers. In *Proceedings of the Working Conference on Parallel Architectures and Compilation Techniques*, August 1994. To appear.
- [CF87] R. Cytron and J. Ferrante. What's in a name? In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19-27, August 1987.
- [CFR⁺91] R. Cytron, J. Ferrente, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451-490, October 1991.
- [Ebc88] K. Ebcioglu. Some design ideas for a vliw architecture for sequential-natured software. In *IFIP Proceedings*, 1988.
- [Gir91] M. Girkar. *Functional Parallelism: Theoretical Foundations and Implementations*. PhD thesis, University of Illinois at Urbana-Champaign, 1991.
- [GP92] M. Girkar and C.D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):166-178, March 1992.
- [GS90] R. Gupta and M. L. Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *Transactions on Software Engineering*, 16(4), April 1990.
- [KMC72] D. J. Kuck, Y. Muraoka, and S. C. Chen. On the number of operations simultaneously executable in fortran-like programs and their resulting speedup. *IEEE Transactions on Computers*, December 1972.
- [Mas87] H. Massalin. Superoptimizer: A look at the smallest program. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, 1987.

- [ME92] S. Moon and K. Ebcioglu. An efficient resource constrained global scheduling technique for superscalar and vliw processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, Portland, OR, December 1992.
- [NE90] T. Nakatani and K. Ebcioglu. Using a lookahead window in a compaction-based parallelizing compiler. In *Proceedings of the 23rd Annual International Symposium on Microarchitecture*, 1990.
- [Nic85] A. Nicolau. Uniform parallelism exploitation in ordinary programs. In *Proceedings of the 1985 International Conference on Parallel Processing*, 1985.
- [NN92a] S. Novack and A. Nicolau. An efficient global resource constrained technique for exploiting instruction level parallelism. In *Proceedings of the 1992 International Conference on Parallel Processing*, St. Charles, IL, August 1992.
- [NN92b] S. Novack and A. Nicolau. Trailblazing: A hierarchical approach to percolation scheduling. Technical Report TR-92-56, University of California at Irvine, 1992. Also appears in the Proceedings of the 1993 International Conference on Parallel Processing, St. Charles, IL., August 1993.
- [NN94] S. Novack and A. Nicolau. Resource-directed loop pipelining. Technical report, Univ. of Calif., Irvine, Dept. of Information and Computer Science, 1994.
- [NP91] A. Nicolau and R. Potasman. Incremental tree height reduction for high level synthesis. In *Proceedings of the 28th ACM IEEE Design Automation Conference*, San Francisco, CA, June 1991.
- [NPW91] A. Nicolau, R. Potasman, and H. Wang. Register allocation, renaming and their impact on parallelism. In *Proceedings of the 4th International Workshop on Languages and Compilers for Parallel Processing*, 1991.
- [Pot91] R. Potasman. *Percolation-Based Compiling for Evaluation of Parallelism and Hardware Design Trade-Offs*. PhD thesis, University of California at Irvine, 1991.
- [RG82] B.R. Rau and C.D. Glaeser. Efficient code generation for horizontal architectures: Compiler techniques and architectural support. In *Proceedings of the 9th Symposium on Computer Architecture*, April 1982.