RC 20495 (7/9/96)
Computer Sciences/Mathematics

IBM Research Report

Architecture, compiler and simulation of a tree-based VLIW processor

J.H. Moreno, M. Moudgill, K. Ebcioglu, E. Altman,
B. Hall,* R. Miranda, S.K. Chen, A. Polyak

IBM Research Division
T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

*On assignment from IBM Toronto, Toronto, Canada

Architecture, compiler and simulation of a tree-based VLIW processor*

J.H. Moreno, M. Moudgill, K. Ebcioglu,
E. Altman, B. Hall, ** **R. Miranda,**
S.K. Chen, A. Polyak

IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218, Yorktown Heights, NY 10598

Abstract

We describe a processor architecture based on *very-long instruction word (VLIW)* principles, its associated optimizing compiler, the evaluation approach used to measure its potential performance, and the simulation environment developed for experimenting and performing trade-offs analysis among alternative features of architecture and compiler.

In this architecture, programs are encoded as sequences of *tree-instructions* which do not explicitly reflect the organization of the processor where they are executed. Tree-instructions are dynamically translated, during instruction cache reloading/accessing, into an implementation-specific VLIW form that preserves the simple instruction-dispatch logic characterizing VLIW processors. Large tree-instructions are dynamically *pruned* to fit into smaller implementations. This scheme makes possible object-code compatibility between VLIW processors with different organizations as well as with scalar/superscalar implementations, thus solving one of the most critical objections to the adoption of the VLIW paradigm in general-purpose architectures.

The compiler uses state-of-the-art optimizing techniques to reach new levels of sustained instruction-level parallelism, in branch-intensive and numeric-intensive programs, while also exhibiting much flexibility so that architecture/compiler/implementation trade-offs can be explored in several dimensions. The simulation environment integrates a collection of innovative and known techniques to deliver fast turn-around time for preliminary performance estimates, so that architecture/compiler trade-offs can be analyzed over complete execution runs, as well as complete cycle-by-cycle tracing and timing information.

* We thank and acknowledge the contributions of Norman Cohen, Richard Goldberg, Peter Oden, Vladimir Kotlyar, Induprakash Kodukula and Balaram Sinharoy.

** On assignment from IBM Toronto, Toronto, Canada

1. Introduction*

The design of a new computer architecture, or a new implementation of an existing architecture, is a complex process. For a given set of requirements, designers must

- determine what attributes are important and necessary;
- design a machine implementing such attributes, which delivers adequate performance with respect to other architectures/implementations;
- fulfill implementation and cost constraints.

Moreover, the requirements are not always completely defined in advance and/or fixed. The expected applications (software) as well as the expected technology (hardware) may change dramatically from the time of conception to the time of delivery of a new machine. Consequently, a new architecture or implementation must be designed under various assumptions regarding the features of software and hardware.

Due to the overall complexity of the process, designers usually select a subset of the issues involved and perform trade-offs only among those issues. For example, the design of a new implementation (a new micro-architecture) includes trade-offs regarding functional organization, logic and circuit design; some compiler functionality might be included in such a design, though frequently it seems that the compiler effects are taken into account after the micro-architecture has been developed. In contrast, the typical case for a new computer architecture consists of focusing first on instruction set, machine organization and compiler functionality, while making assumptions regarding implementation technology, chip size and so on.

The evaluation of trade-offs such as the ones mentioned above requires adequate tools for simulation and performance measurement [1]. In general, the design process is an iterative one: features are proposed, required changes are introduced to the design environment, correctness of the changes is verified, and performance of the new features is evaluated. Such a process is required due to the ever tighter interaction among the different elements in a design. Consequently, the tools used should be efficient, allow experimentation with realistic work-loads, be easy to reconfigure, and permit the evaluation of a variety of system design issues. Moreover, the tools should be able to support the development of the target system as well as the performance evaluation process, activities which have widely different requirements in terms of their acceptable turn-around time and the accuracy expected from the results.

Instruction-level parallelism

An important focus of activity regarding computer architectures/micro-architectures in recent years has been the exploitation of instruction-level parallelism (ILP), that is, the ability to simultaneously execute several operations [2-3]. Processors capable of exploiting ILP contain multiple functional units, fetch several instructions per cycle from the instruction cache, and in a given cycle may dispatch multiple operations for execution. Such processors are referred to as *superscalar*, to make the distinction with *scalar* processors which dispatch at most one instruction per cycle [2]. Moreover, when the set of operations issued for parallel execution is large (say larger than eight), the implementations are also referred to as *wide-issue superscalar processors.*

Depending on the machine organization, the multiple simultaneous operations executed by a superscalar processor may be an *in-order* or *out-of-order* sequence of instructions. In the case of an in-order sequence, the corresponding processor has some mechanism to determine the set of adjacent operations that are executable simultaneously, and issues them in the order in which they appear in the program. In contrast, an out-of-order processor has hardware resources for determining the dependencies among successive operations as well as the order in which those operations are executed; that ordering may be different from the

* The following terms used in this paper are trademark of International Business Machines Corporation: RS/6000, PowerPC, AIX.

order in which instructions appear in the program. Most existing modern (high-performance) processor implementations belong to the out-of-order class [4].

The instructions executed simultaneously by an in-order issue processor are determined by the way that instructions are placed in a program (that is, by the compiler or programmer), whereas an out-of-order processor may discover those instructions while the program is being executed. Consequently, an in-order processor requires simpler dispatch logic than an out-of-order processor, potentially leading to a faster implementation (higher clock frequency) and/or a shorter design cycle. This is one example of the many trade-offs possible in the design of a processor.

Recent advances in compiler technology, as well as the complexity arising from hardware reordering for wide-issue machines, have been advocating the adoption of in-order architectures together with aggressive compiler optimization techniques (such as scheduling code speculatively across basic blocks, software pipelining, loop unrolling) to group independent instructions executable in parallel [3].

VLIW processors

Aggressive in-order wide-issue execution leads to *very-long instruction word (VLIW)* processors, wherein scheduling of instructions is done statically by the compiler/programmer [3]. VLIW is not a recent concept; in fact, it originated as an extension of microcode-based techniques [5-6], though the concept has changed drastically since its inception.

As with all other processors, VLIW-based processors are subject to a large variety of trade-offs. On one end, VLIW may be regarded simply as an alternative implementation of an existing sequential architecture, in which case the architecture is fixed but the compiler is augmented with the functionality for generating VLIWs (i.e., grouping together instructions executable in parallel); perhaps some architecture modifications might be introduced to simplify the process of detecting at runtime the boundaries of VLIWs in a program. On the other end, VLIW may be regarded as a completely new architecture, in which case the formats and contents of VLIWs can be defined in a manner different from a sequential program representation leading to simpler decoding and issuing of instructions.

VLIW implementations have been described as “a natural successor to RISC,” because complexity is moved from the hardware into the compiler. As stated in [7], “the objective of VLIW is to eliminate the complicated instruction scheduling and parallel dispatch that occurs in most modern microprocessors. In theory, a VLIW processor should be faster and less expensive than a comparable RISC processor.” The transfer of complexity from hardware into the compiler is another source for trade-offs in a VLIW-based design.

In spite of the attractiveness of its potential design simplicity, VLIW implementations have been perceived as suffering from important limitations when compared to out-of-order implementations, such as the need for a powerful compiler, larger code size arising from the aggressive scheduling performed by the compiler, lock-step execution of VLIWs, and binary incompatibility across implementations with varying number of functional units [2] [7] [13].

Scope of the paper

We have been researching the viability of a VLIW architecture in the context of the AIX operating system. Our focus has been on the development of a new architecture, with enhancements for exploiting ILP such as the removal of unique resources like the carry bit (which may cause serializations), or the availability of a larger register set. We have been studying the potential features of such a new architecture, the appropriate compiler algorithms, and the interaction among architecture and compiler. The objective has been the development of an architecture/compiler combination which reaches new levels of sustained ILP for branch-intensive programs and for numeric-intensive programs.

Our research is oriented towards the evaluation of trade-offs for a VLIW architecture/compiler combination. Consequently, we need a suitable design process and simulation environment for experimenting with alternative features and for evaluating the benefits and complexities arising from such alternatives. In this paper, we describe a process and environment providing reasonably *fast turn-around time from compilation to simulation*, at different levels of accuracy regarding performance estimates, so that *architecture/compiler trade-offs* as well as *implementation trade-offs* can be analyzed over complete execution runs.

Our tools have similar properties to those available in other simulation environments [1], but a combination of features make ours unique, including:

- a highly modular organization;
- fast turn-around time from simulation;
- the integration of the different components, in particular the integration of the simulator, the generation of traces, and the trace-driven timing analysis;
- the capability of timing the complete execution of programs without the need for storing traces;
- the capability to obtain different levels of accuracy in the performance measures, with more accurate results requiring longer execution time;
- the applicability to any type of processor architecture or microarchitecture, but particularly well suited to modeling wide-issue processors such as VLIW due to lower simulation overhead.

Attempting to illustrate the similarities/differences among other tools and ours is out of the scope of this paper; instead, we just relate some relevant features of our environment with those provided by Shade [8].

We first review the issue of object-code compatibility in VLIW processors, then summarize the relevant features of our proposed *tree-based VLIW architecture*, describe the significant aspects of the associated *optimizing compiler*, review the *iterative design process* and the *simulation environment* developed for experimenting with alternative features, and finally describe some of the types of trade-offs possible. Quantitative results obtained from the experiments carried out -measured ILP and speedup with respect to scalar implementations, impact of different alternatives, and so on- are being reported elsewhere [9].

2. Object-code compatibility in VLIW processors

Modern computer architectures exhibit object-code compatibility, that is, the capability to execute the same code on all implementations of the architecture. Although members of a given processor family may differ in their resources and capabilities for dispatching primitive instructions, and object code may be tuned for the resources available in a specific implementation, all processors can execute the same object code.

VLIW object-code such as that used in [10-12] is explicitly dependent on the features of the implementation (i.e., the number, type and location of the functional units). This has been a conscious decision, under the assumptions that the implementations would be simpler, and the compiler could better exploit the processor resources if it had good knowledge of the corresponding features and limitations. In other words, the separation among *architecture* and *implementation* that has been common practice in processor design for scalar/superscalar implementations has been abandoned in VLIW implementations, in order to simplify and exploit better the hardware by the compiler/programmer.

The drawback of VLIW as described above is that it sacrifices object-code compatibility among VLIW implementations and with sequential (scalar/superscalar) implementations. The VLIW code explicitly indicates the detailed structure of one specific implementation (number and type of functional units) which is different from others, and/or the VLIW code uses features that might not exist in sequential implementations. The cost associated with porting and maintaining code across implementations has limited the acceptability of the VLIW paradigm as part of a processor architecture [2] [7] [13].

Two classes of solutions have been pursued to solve the VLIW object-code compatibility problem: software-based and hardware-based techniques. Software-based approaches (excluding source recompiling due to its inherent limitations) rely on binary-to-binary (object-code) translation, either with or without hardware support [14-17]; some of these actually correspond to translating binary code among different architectures, but they can be applied to different implementations of the same architecture, including VLIW-based ones.

Of particular relevance is the scheme described in [17], which addresses the translation of code from one VLIW implementation to another by performing code rescheduling at page-fault time. This is a promising general technique, capable of coping with any implementation constraint; its limitations are the overhead introduced at page-fault time and the added complexity required to manage rescheduled and non-rescheduled pages. In contrast to earlier solutions, the scheme assumes that the object-code is already an explicit representation of ILP, and is able to exploit that representation accordingly.

A hardware-based solution for VLIW processors was proposed in [18], wherein the concept of *delayed split-issue* together with dynamic scheduling hardware are introduced; these characteristics allow using the interlocking and score-boarding techniques known for dynamic scheduling in superscalar processors. In fact, the primary objective of that paper is "to show that dynamic scheduling is as viable for VLIW processors as with more conventional ones." As a result, the object-code generated for one implementation of a VLIW processor family can be executed on an implementation with fewer functional units and/or different latencies for the operations.

Another related class of hardware-based solutions consists of using sequential code as the compatible representation and performing dynamic parallelizing. For example, the *fill-unit* approach originally proposed in [19] and extended in [20] extracts VLIWs from the sequential execution of a program, so that successive executions of the same instructions are performed using the parallel rather than the original sequential version. However, this type of approaches is limited due to the small "window" of instructions analyzed, and the complexity associated to run-time parallelizing.

Since there have been very few proposals on representing a VLIW program without depending on the specific aspects of an implementation, it appears that exposing the details of a VLIW implementation to the compiler/programmer also *requires* that such details be reflected in the object code. We argue that this is not necessary, in spite of the perception to the contrary; in fact, the architecture described in the next section provides a solution to this problem.

3. A VLIW architecture based on tree-instructions

Let us address the features of our proposed VLIW architecture. In this architecture, a program consists of a sequence of *tree-instructions*, or simply *trees*, each of which corresponds to an *unlimited multiway branch* with multiple branch targets and an *unlimited set of primitive operations* (see Figure 1). All operations and branches are independent and executable in parallel, as a single VLIW. The multiway branch is associated with the internal nodes of the tree, whereas the operations are associated with the arcs. The multiway branch is the result of a set of *binary tests on condition registers*: the left outgoing arc from a tree node corresponds to the false outcome of the associated test, and the right outgoing arc corresponds to its true outcome. These tree-instructions are an extension of those described in [21-22].

Based on the evaluation of the multiway branch, a single path within a tree-instruction is selected at execution time as the *taken path* (a tree-path starts from the root of the tree and ends in a branch target). Operations on the taken path are executed to completion, and their results placed in the corresponding target registers or storage locations. In contrast, operations not on the taken path of the multiway-branch are inhibited from committing their results to storage or registers. Such operations do not produce any effect on the architectural state of the processor.

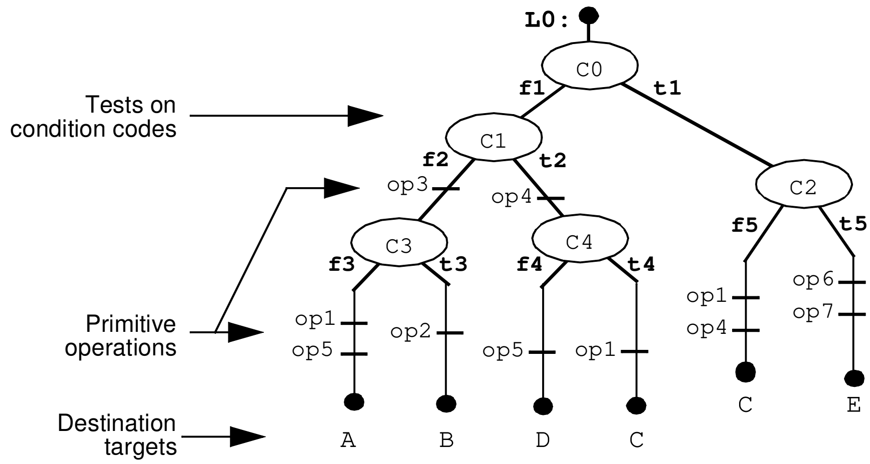

Figure 1: A tree-instruction

Constraints in tree-paths

The primitive operations in a tree are subject to *sequential constraints for each path of the tree.* That is, the final result from executing a tree must be the same as if all primitive operations in the taken path are executed one at a time (sequentially). Consequently, a primitive operation *cannot* use a resource which has been set by a previous operation in the same tree-path (a read after write conflict); Figure 2 illustrates legal and illegal tree-instructions. If this constraint is not fulfilled within a tree-instruction, the results from primitive operations having an operand set by a previous operation in the same tree-path are undefined.

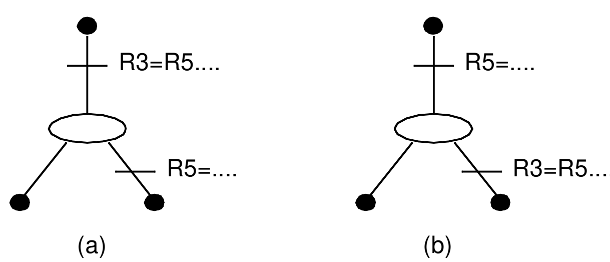

Figure 2: Legal (a) and illegal (b) tree-instructions

Memory representation of a tree-instruction

Each tree-instruction is represented in main storage as a *contiguous sequence of tree-chunks* (or simply *chunks),* with each chunk containing the primitive instructions. A chunk is the minimum unit of program specification, and the size of a tree-instruction is a multiple of the size of a chunk. For example, a chunk could correspond just to a single memory word and each primitive operation would be encoded in one memory word; alternatively, a chunk could correspond to four memory words (a quad-word) and several operations could be encoded in such a chunk. The chunk size determines trade-offs in memory space and decoding complexity, but it is fixed for a particular architecture. For simplicity, in the remainder of this paper we assume one-word chunks; additional details regarding multiword chunks are described in [23].

Each tree-instruction is represented in main storage as a *contiguous sequence of primitive instructions* which is obtained from the *depth-first traversal* of a tree-instruction. For example, the code sequence in Figure 3 represents the 6-way tree-instruction depicted in Figure 1; note that, in contrast to traditional VLIW processors, this representation does not use no-ops in the encoding of programs.

```
L0: skip C0,t1          t4: op1
f1: skip C1,t2              branch C
f2: op3                 t1: skip C2,t5
    skip C3,t3          f5: op1
f3: op1                     op4
    op5                     branch C
    branch A            t5: op6
t3: op2                     op7
    branch B                branch E
t2: op4
    skip C4,t4          L1: ...
f4: op5
    branch D
```

Figure 3: Sequential representation of a tree-instruction

Testing a condition register is performed with a *skip conditional* instruction, which corresponds to a flow-control operation within the tree, and which indicates where the tree-path corresponding to the true outcome of the test continues in storage; as a result, a skip conditional instruction is a branch with a (short) positive displacement. On the other hand, the branch targets of a tree are represented as *unconditional branch instructions,* which specify the next tree to be executed when the corresponding path of the current tree is selected.

The program representation illustrated in Figure 3 is directly executable by a superscalar processor, which can use hardware dependence-checking and scheduling resources to determine which operations can be executed simultaneously. In the case of a VLIW processor, the *end of a tree-instruction* is delimited by a primitive instruction that follows an unconditional branch which is not reachable by any skip conditional instruction within the tree. In the tree-instruction above, for example, the end of the tree is detected at label `L1` because such a label does not appear in any of the skip instructions.

Note that each primitive instruction within the tree-instruction in Figure 3 may correspond to the starting point of a different tree. As a result, branching into a tree-instruction leads to the execution of a tree which is a subset of a larger tree. For example, branching into the instruction at label `f2` in Figure 3 leads to the execution of the two-way tree instruction ending at `branch B`.

Execution of a tree-instruction in a processor with few resources

Sequential constraints in each tree-path make possible *binary compatibility among different implementations* of this architecture, each with varying degrees of parallel execution capabilities, by allowing the decomposition of a large tree into subtrees which are executed in different cycles. If the size of a tree-instruction exceeds the resources available in a processor implementation (such as branching degree, number of fixed-point or floating-point operations), then the tree-instruction is dynamically decomposed *(pruned)* to fit the resources. The resulting subtrees have the same general structure as the original trees (that is, a multiway-branch tree with operations in the tree-paths), but their size is limited. Subtrees are executed in successive cycles, unless the taken path is completely contained within the first subtree.

As an example of the pruning process, consider the execution of the tree-instruction depicted in Figure 1 by a VLIW processor capable of executing at most a 4-way branch. Since the tree-instruction contains a 6-way branch, it must be pruned. In this case, pruning takes place at the first two skip instructions, as depicted in Figure 4. As illustrated, the original tree-instruction is decomposed into three smaller subtrees, the first one containing a 4-way branch (targets T1,T2,A,B), and the other two containing 2-way branches (targets D,C and C,E, respectively). A similar decomposition occurs whenever a tree-instruction exceeds any other type of resource (such as arithmetic-logic or load/store units).

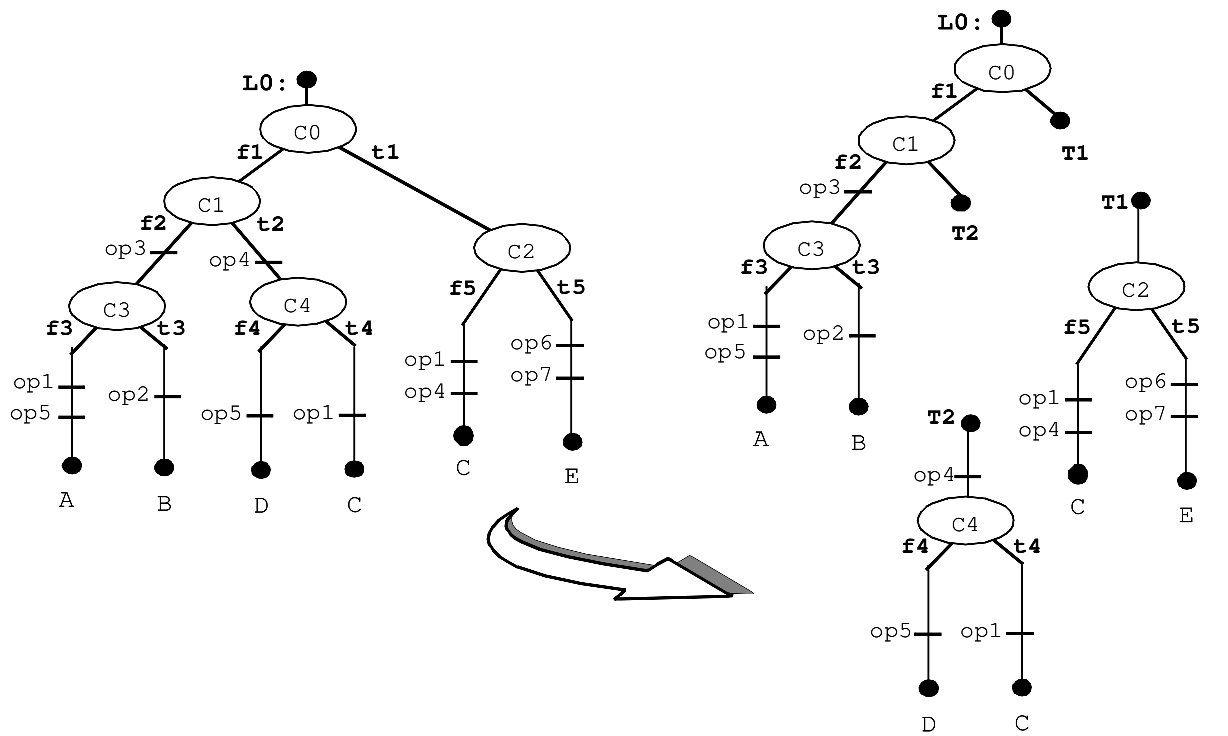

Figure 4: Pruning a tree-instruction

Performance implications of pruning

One objective of this architecture is providing scalable performance at the object-code level, that is, the same object-code should deliver better performance in larger implementations. For this purpose, the architecture assumes that a compiler/programmer generates tree-instructions targeted for the largest implementation available, or even larger. That is, the object-code may contain as much instruction-level parallelism as is possible to extract from a program, including speculative and conditional instructions, while introducing minor degradation with respect to code recompiled for an specific implementation.

Execution paths are assigned to tree paths in left-to-right order, according to their probability of being taken: the most probable path is the left-most one whereas the least probable path is the right-most one. As a result, all operations in the most probable path appear in adjacent locations in memory (from `L0` up to `branch A` in Figure 3). Operations are placed on paths other than the left-most one to take advantage of available resources, if they exist.

For implementations with fewer resources than those assumed available when a program was generated, the pruning process automatically eliminates some of the conditional operations (the right-most parts of the trees), if they exist. Consequently, pruning reduces the performance of a program when the number of operations in the taken path exceeds the resources available, or when the taken path is not the most probable one. On the other hand, as long as the taken path is fully contained within the first subtree, no performance degradation occurs because the remaining subtrees are not accessed at all.

However, if a program is generated assuming the availability of k functional units but is executed in a processor implementation with $l > k$ units, then there is neither performance gain nor performance degradation with respect to execution on an implementation with exactly k units.

4. Translation of tree-instructions

As it has already been stated, tree-instructions are dynamically translated into a VLIW form; this is accomplished in a two-step process, as follows:

Pruning step. At instruction cache (I-cache) reload time, trees are decomposed into subtrees whose maximum size fits the capabilities of the VLIW processor, as described in the previous section. Information extracted from the trees is added in the I-cache, identifying the starting and ending position of subtrees within each cache line, the number and type of operations per subtree, and so on.

Alignment step. At I-cache access time, the subtree starting at the specified memory address is extracted from the corresponding I-cache line and is aligned to match the resources of the VLIW processor. The alignment is achieved using the extra information encoded in the same I-cache line.

Consequently, VLIW-based implementations of this architecture are different from classical ones, because the instruction cache does not contain VLIWs; instead, the predecoded information added in the I-cache is used to format the VLIWs as they are fetched from the cache. On the other hand, these implementations fetch exactly as many primitive instructions as the maximum that the processor is capable of executing at once (i.e., a VLIW). If the subtree to be executed contains fewer operations than those fetched from the I-cache, then the alignment step disables those operations beyond the end of the subtree (as if they were a path that is never taken).

The pruning algorithm, performed at I-cache reloading, is described by the pseudo-code depicted in Figure 5 [23]. Basically, the procedure consists of fetching a block from main memory and inspecting each primitive instruction in the block. For each instruction, it is added into the current subtree if there are resources available of the corresponding type; otherwise, the tree-instruction is pruned at that point, completing the current subtree. If no pruning occurs, the current subtree is completed when the number of unconditional branch instructions matches the number of paths as determined by the number of skip instructions (each skip instruction adds a path).

```
fetch memory block;
foreach (instruction) loop
  increment_count_type(instruction);
  if    (num_skips  > max_skips)  then prune();
  elsif (num_classA > max_classA) then prune();
  ...
  elsif (num_branches = num_paths) then emit_subtree();
  else
      insert(instruction);
  end if;
end loop;
```

Figure 5: Pruning algorithm

Figure 6 illustrates the outcome of the pruning step for the tree-instruction listed in Figure 3, assuming a processor capable of executing a 4-way branch and 4 primitive operations per VLIW; for simplicity, we assume a large I-cache line. As depicted in Figure 6, the original tree is decomposed into four subtrees, indicated by the heavy lines in the figure. Subtree labeled `L0` contains four branches and three operations; subtree labeled `T3` consists of just one operation and a branch; subtree `T2` specifies two branches, one skip instruction and three operations; and subtree labeled `T1` contains one skip, two branches, and four basic operations. Additional bits (shown below the instructions in Figure 6) specify the boundaries of subtrees within the I-cache line, as well as the location of primitive operations, pruning points, and branches within the lines.

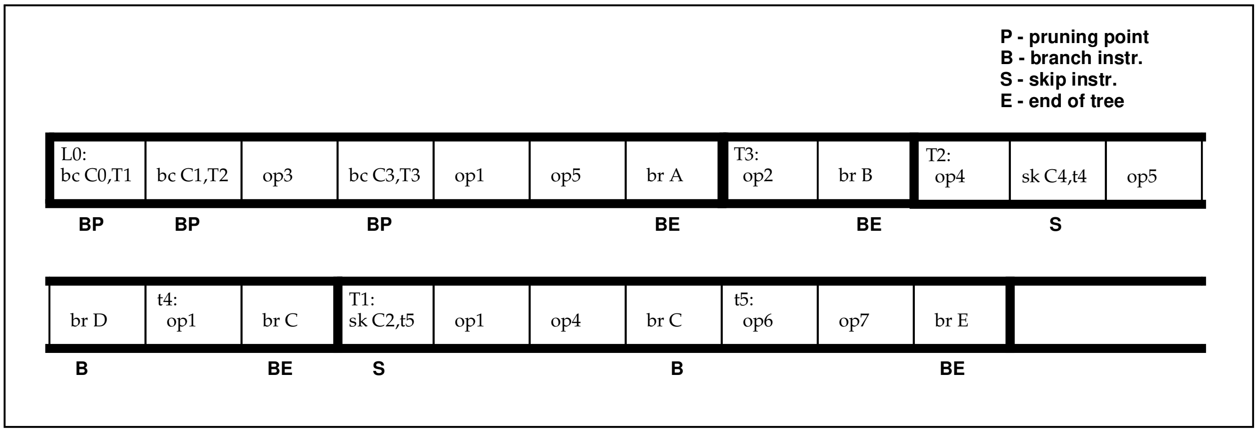

Figure 6: Subtrees in instruction cache

A simple implementation of the pruning step is a sequential one, in which one primitive instruction is examined at a time and the actions indicated in Figure 5 are performed. The detection of the end of a tree, as well as the pruning points, can be effectively supported by a stack of tree-paths: each skip instruction pushes the current path into the stack, and each unconditional branch instruction pops the previous uncompleted path from the stack. The end of the tree-instruction corresponds to an empty stack, whereas pruning points correspond to the paths remaining in the stack when a processor resource is exhausted. A parallel implementation is described in [23].

5. Processor organization

Figure 7 depicts the overall organization of a VLIW-based implementation of this architecture. Tree-instructions are fetched from main memory and fed into a *pruning unit,* which converts them into subtrees whose requirements fit the characteristics of the specific processor implementation; the subtrees resulting from the pruning process are stored into the *instruction cache.* A VLIW fetched from the instruction cache is aligned by the *alignment unit* and placed in the *VLIW Register*, which feeds the *multiway branch unit* and multiple *fixed-point* and *floating-point units.* The branch unit generates the address for the next tree-instruction, whereas the fixed-point and floating-point units perform the operations and interact with register files and storage.

Single-cycle multiway branching

The proposed architecture relies on multiway branching in every cycle to handle branch-intensive programs as well as to exploit conditional execution of operations. Branch evaluation is based on the status of the condition register prior to the VLIW being executed, so there are no intra-VLIW dependencies in the branch evaluation. In spite of that, single-cycle multiway branching is a quite stringent requirement because it relies on selecting and fetching one out of several possible next tree-instructions in every cycle.

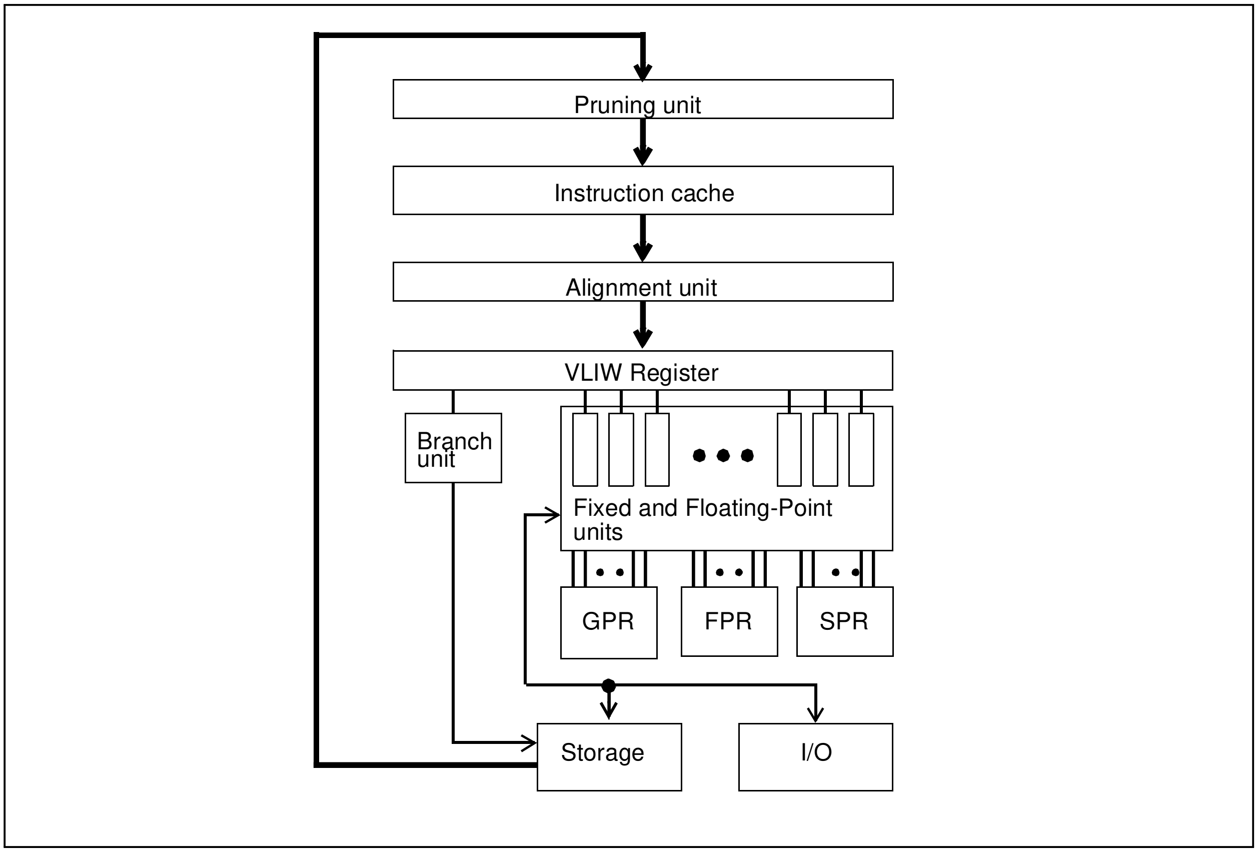

Figure 7: VLIW-based processor organization

The mechanism used to timely fetch the next tree-instruction from storage consists of allocating all possible next tree-instructions for a given tree as a block in adjacent memory location. A block is prefetch from the instruction cache by using the high-order address bits of the first target (which are the same for all targets). The actual target VLIW is extracted from the block as soon as the taken path through the tree becomes known.

Allocating all possible next tree-instructions as a block leads to replication of tree-instructions, whenever a given tree is the target of two or more trees and is not possible to arrange them so that they are all allocated in the same memory block. Thus, this approach leads to larger instruction memory requirements due to code replication. Packing tree-instructions into blocks, including the potential replication, is performed by the VLIW assembler.

Semantics of a VLIW

As already stated, primitive instructions on the taken path of the multiway-branch are executed to completion and their results placed in the corresponding target registers or storage locations. The following rules apply to these instructions:

- all these instructions are executed concurrently;
- the results from an operation that uses a processor resource set by a previous operation in the path are undefined, with the exception indicated next; and
- if two or more *store* instructions target the same memory byte, the value stored in that byte corresponds to the store instruction appearing later in the tree-instruction (in sequential storage order).

Register set

In general, the size of the register set in our VLIW architecture is larger than the one available in the PowerPC architecture, though we have been experimenting with a range of sizes; there is a general-purpose (integer) register set, a floating-point register set, and a collection of special-purpose registers. Moreover, some special-purpose registers are replicated so that conflicts arising from contention for a single resource have been eliminated. For example, there are multiple fixed-point exception registers (known as XER in the PowerPC architecture) [31], so that multiple "add carrying" operations can be performed in parallel (the "add carrying" instruction sets bit CA in XER (the *carry* bit); in our VLIW architecture, the "add carrying" instruction specifies a target XER in addition to the target general-purpose register). The same approach is used with those instructions that use CA as an operand (such as "add extended" instructions).

Primitive instructions

The primitive instruction set in the proposed VLIW architecture follows closely the instruction set in the PowerPC architecture, but there are a number of important modifications. Some complex (multi-cycle) instructions, such as *load/store multiple* and *load string*, have been replaced by multiple single-cycle operations; we have also evaluated the decomposition of *load/store with update* instructions into separate primitives: *load/store, add*. These modifications allow for simpler scheduling algorithms in the compiler, as well as simpler implementations.

On the other hand, we have also evaluated the introduction of some instructions which are more complex than those available in the PowerPC architecture; these include, for example, instructions to perform pairs of simple but frequent operations such as *shift-add* and *add-store*.

Speculative execution and out-of-order memory operations

Speculative execution and out-of-order memory references are techniques used by the parallelizing compiler for extracting instruction-level parallelism [2-3]. A *speculative operation* is one that has been placed above a branch with respect to a sequential execution stream, on the *speculation* that the result will be needed. If subsequent events indicate that the speculative instruction would not have been executed, or the results of the speculative instruction are not valid, any result produced by the instruction is not used. Typically, instructions are placed speculatively by the compiler when there are resources that would otherwise be idle so that the operation is done without cost, or to reduce delays in the program. Most fixed-point instructions (arithmetic, logic) can be executed speculatively. Store instructions should not be executed speculatively, nor should other instructions that produce unrecoverable effects.

An *out-of-order load* instruction is one that has been placed above a store instruction with respect to sequential execution. Load instructions frequently start a sequence of dependent operations that depend on the datum loaded, so it is advantageous to initiate the loads as early as possible. However, an out-of-order load may conflict with a store operation over which it has been moved if the addresses referenced cannot be disambiguated by the compiler.

The instruction set in our VLIW architecture includes support for *speculative instructions* and for *out-of-order memory operations*. The support consists of a *bit in each register* to indicate a speculative value, a *bit in load instructions* which indicates whether an instruction is speculative, *address comparators* to detect overlapping memory references, and special *commit* instructions which remove the speculative status of a value [24]. If an exception is encountered during execution of an speculative instruction, such an exception is recorded but not generated until reaching the corresponding commit instruction. All these mechanisms are exploited by the compiler to perform aggressive code motions in the program.

6. The VLIW evaluation compiler

We now address the features of our VLIW compiler. The current version of this compiler is the third one written as part of our research activities, so that many of its features are based on our own experience in the area. The first two efforts were really intended as binary-to-binary translators, which would accept RS/6000 object files and produce VLIW assembly files. In contrast, the current version is an optimizing back-end that exposes instruction-level parallelism. This compiler can be used to generate code for any machine that exploits large amounts of ILP, with the primary target being a VLIW-based processor capable of issuing many instructions at a time (either 8, 16 or more, depending on the number of resources in the target implementation). It is the responsibility of the compiler to find enough instructions to exploit the available resources.

Unlike for many other compilers, the target architecture is not fixed. One use of the compiler is to investigate trade-offs among resources such as the size of the register set and the number of execution units, so the compiler must be able to handle different processor configurations. Moreover, the compiler must be able to support the evaluation of alternative features in the architecture by making possible to determine the performance of compiled code with and without an additional feature.

In summary, the requirements for the compiler are

- ability to generate optimized code to properly exploit the VLIW hardware;
- ability to support multiple processor configurations, so that the number and balance of various resources can be studied; and
- modifiability to allow exploring novel compiler optimizations and hardware ideas.

Supporting architectural explorations and implementing aggressive optimizations geared towards several different targets implies a compiler in constant state of change. Consequently, the compiler design must support adding code, verifying the modified compiler, and isolating problems quickly. Such mutability is governed by the requirement for stability.

Parallelization is accomplished using the techniques originally described in [22]: given sequential code as input, independent instructions are identified (and in some cases, created) through various compiler optimizations, and these instructions are grouped together by the scheduler. When the target is a VLIW machine as the one described in the previous section, the parallel groups are expressed as tree-instructions (see Figure 8). The transformations in the compiler can be roughly divided into the categories depicted in Figure 9.

6.1 Extracting instruction-level parallelism

The main compiler optimizations are geared towards exposing, increasing and exploiting instruction-level parallelism. Other optimizations include traditional transformations [29], and those developed to support specific hardware features.

Software pipelining

Instruction-level parallelism is exploited by scheduling the instructions in such a manner that independent instructions can be executed simultaneously. Our scheduler is built around *software pipelining*, using an enhanced version of the algorithm described in [21-22]. The enhancements, which yield greatly increased ILP, include:

- *height heuristics:* the earlier algorithm schedules instructions greedily - i.e. an instruction is scheduled as soon as possible. Instead, our algorithm uses heuristics to delay the scheduling of an instruction, resulting in better schedules.

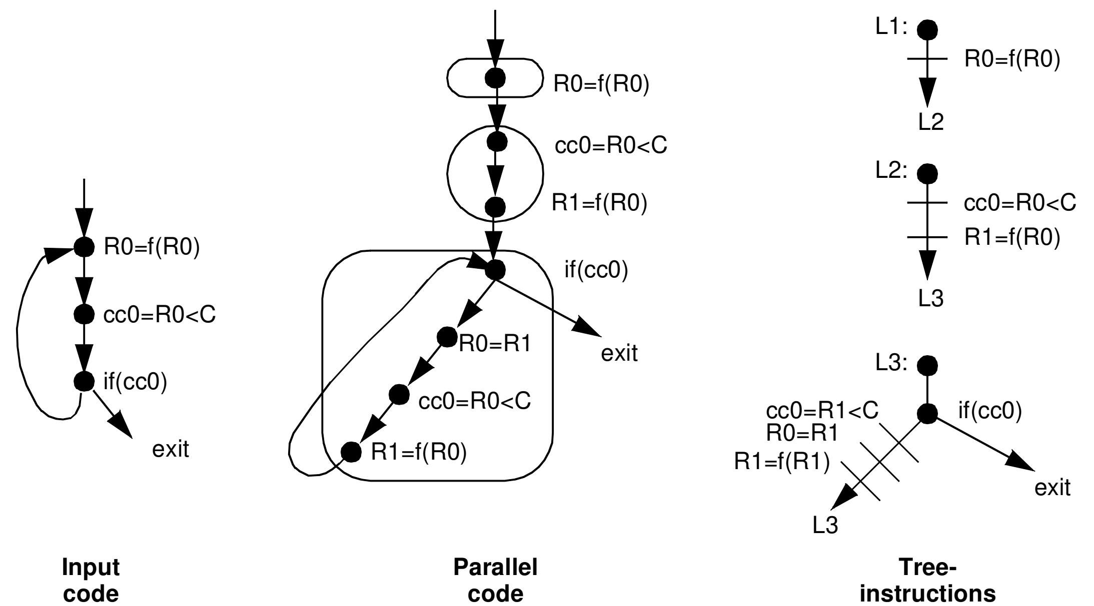

Figure 8: The parallelization process

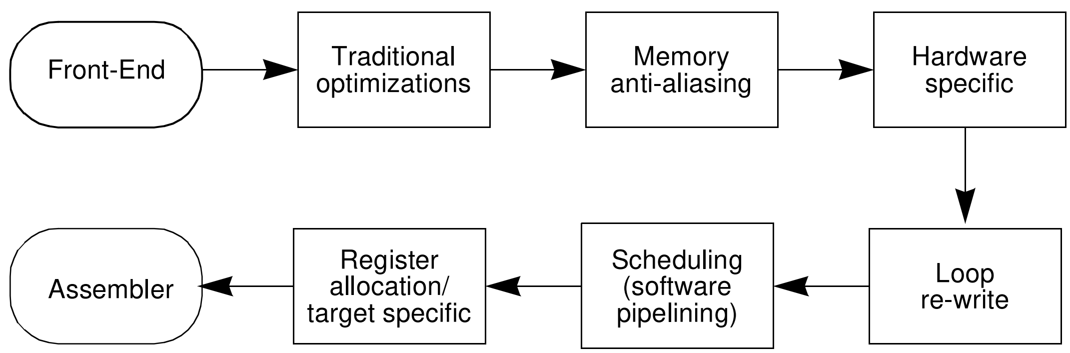

Figure 9: Phases in the compiler

- *infinite scheduling window:* the earlier algorithm looks at a restricted window of instructions in a loop, typically 64 to 128, due to compilation time concerns. We have recast the algorithm so that it examines entire loops, without increasing compilation time significantly.

Software-pipelining is applied pervasively, to outer as well as inner loops; it is also used on loops with complex bodies and loops with multiple heads.

Memory disambiguation

A lot of effort in exposing parallelism is devoted to memory disambiguation. Serialization among memory operations due to potential memory aliases causes a large decrease in the amount of ILP available. Some of the techniques we use are:

- *traditional address disambiguation:* we use aggressive versions of traditional techniques [29]. We compute whole-function symbolic expressions for addresses, which are then used to compute aliases among adjacent load/store pairs. The disambiguation techniques also include the use of Banerjee and GCD tests across loop iterations [30].
- *loop cloning:* loops may contain a dependence which forces a mostly sequential execution. However, in many cases it is possible to determine, by checking a simple condition outside the loop, that no dependence exists and the loop can be parallelized. If there is such a condition, we *clone* (duplicate) the loop; one version contains the dependence and its execution is mostly sequential, whereas the other is parallelized. At run-time, the parallel version of the loop is executed whenever the condition is satisfied, greatly increasing instruction-level parallelism.
- *address comparators:* this is a technique that relies on hardware-support to allow moving a load operation earlier than a preceding store operation in the sequential execution stream, even if it cannot be guaranteed that the corresponding memory locations do not overlap [24].

Loop rewriting

Loops, especially in numerical code, contain a lot of potentially parallel instructions. We use several loop-rewrite optimizations [30] to transform loops and increase the amount of ILP available. These transformations include:

- *unrolling loops:* as implemented, software pipelining can start at most one iteration of a loop at a time. Unrolling small loops allows presenting to the scheduler a loop body which contains several iterations of the original loop, so that software pipelining can schedule the initiation of several iterations in the same cycle.
- *reducing minimum-initiation interval* (MII): software-pipelining schedules a loop such that iterations of the loop start once every MII cycles. MII reduction rewrites some loops so that MII is reduced, enabling iterations to be started more frequently.
- *rewriting associative operations:* a sequence of commutative/associative operations can be decomposed into several subsequences, wherein each subsequence can be computed in parallel and the partial results are then combined to get the actual value. Based on this idea, loops containing commutative/associative operations are rewritten so that each iteration of the loop computes several partial results in parallel; this greatly increases ILP on small loops that contain commutative/associative operations.

6.2 Architecture/implementation-specific transformations

The compiler uses parameterization to support multiple targets. Parameters are used to control

- instruction latencies;
- register file sizes;
- the number of execution units of various types; and
- the issue width and mix.

The compiler also supports a variety of optimizations tailored to specific architecture/implementation features. In addition to the address comparators already mentioned, the supported features include:

- *non-trapping/speculative loads,* which enable issuing loads speculatively, thereby removing control dependences that could potentially inhibit performance.
- *conditional instructions,* such as *select* (conditional move) and *conditional store* instructions. These conditional operations do not enhance performance directly; instead, they allow converting if-then-else structures into a mixture of speculative operations and selects thereby enabling the simplification of control flow.
- *combining operations,* such as add-add, add-shift, add-store, into a single operation.

- *prefetch instructions,* also known as *cache touch* instructions, under varying assumptions regarding the memory hierarchy.

6.3 Implementation

As mentioned earlier, exposing high levels of ILP has been a major goal, which requires implementing sophisticated transformations. However, we have also been constrained by other important goals.

One objective of our research has been identifying the proper number and type of functional units, register sets, and other processor resources which deliver better results. In addition, we are also interested in exploring new features of the architecture. These objectives implied that the compiler must not only be parametrizable but it must also allow introducing instructions which capture the basic ideas and optimizations that exploit them as transparently as possible.

Underlying all goals there is an all-pervasive concern: stability; it was necessary to be able to introduce new transformations and/or instructions without affecting the correct behavior of those already in place.

In this section, we briefly describe the implementation of the compiler focusing on those aspects that enabled us to satisfy these other goals.

Front-end

The previous versions of our compiler were designed to provide object-code translation capabilities. The input to those versions was object code produced by *xlc*, the RS/6000 compiler, which was translated into VLIW assembly code. This was a multi-step process: a special tool read the object code and translated it into a textual representation; then, the compiler read the textual form, parallelized it, and emitted another text file which was then assembled or interpreted by the assembler/simulator.

We decided to preserve this process, but with some changes. First, instead of accepting standard output from the *xlc* compiler, we modified its optimizing back-end so that we could intercept it after all standard optimizations have been done but before any scheduling. The resulting file is easier to convert correctly into the textual input form. Moreover, disabling scheduling avoids performing transformations on the code which interfere with the parallelizing optimizations.

The resulting approach has several advantages:

- it exhibits the benefit of all the standard optimizations, without having to recode them (however, some of them had to be reimplemented because they are needed again later in steps performed in our parallelizer);
- the input to the parallelizer was fairly sanitized; and
- it supports all languages whose compiler uses the same backend; this includes C, C++, and FORTRAN.*

Dependence flow graphs

From our previous experience, we knew about the need for a sophisticated internal representation. Intermediate forms like abstract syntax trees, 4-address code, or control flow graphs [29] were not well suited to the kind of analysis and transformations which we are interested in.

We selected a modified version of the dependence-flow graph (DFG) [25] as the intermediate representation. An exhaustive list of the benefits of using the DFG representation is beyond the scope of this report; some of them are:

- DFGs combine control-flow and dependence information into a single coherent form;

* There were some language specific changes required but, at least for C and FORTRAN, they turned out to be minor.

- the dependence-flow information is a fully "pinned" form, that is, it is both in single-static assignment (SSA) and reverse static-single assignment form; and
- DFGs are a fully executable representation.

Analysis is particularly easy with DFGs: SSA/reverse-SSA based analysis requires no extra work. Moreover, the executable nature of the DFG makes attractive the analysis based on partial evaluation techniques. The choice of DFGs is one of the reasons why our implementation of common optimizations has performance advantages compared with traditional implementations.

Another distinct advantage of our intermediate form is the representation of load/store dependences. Memory operations are connected by anti/output/flow dependences; this provides a mechanism whereby an optimization performs memory anti-aliasing/disambiguation once, and then represents the result so that all other optimizations can use it.

The DFG provides almost all the information needed to implement aggressive transformations. In fact, we use only one additional global data-structure: a (pseudo-)interval hierarchy. For reducible graphs, this is the usual interval hierarchy. For irreducible graphs, certain irreducible portions become pseudo-intervals (they have multiple entry points from outside the loop).

Compilation time

The area where our compiler differs most from a production compiler is compilation time, though this is the result of conscious decisions. Whenever there was a trade-off among compilation time and other property such as robustness, maintainability, extensibility, performance or programmer time, we chose against compilation time. For instance, all algorithms are global: entire interval or function, even for very large functions; we use $O(n^2)$ algorithms where necessary, even for large regions. DFGs, while ideal for implementing optimizations, are more memory intensive than other intermediate forms.

We have made extensive use of internal self-checks; we are almost paranoid about it! A non-optimized version of the compiler, with debugging on, can spend up to 2/3 of its execution time in asserts and data-structure integrity validation. There has been a large payoff from this property; when an optimization is implemented incorrectly, it is usually the case that the compiler fails a self-check in the offending optimization rather than producing an incorrectly compiled program. Additionally, even though we use C, we have had no major pointer-related bugs (a minor miracle!).

Other issues

As already stated, the compiler is extensively parameterized; everything from the processor resource model to the instruction set is defined through tables, and are usually modifiable at runtime. For instance, primitive instructions and their properties are in a table; moreover, this tables includes all properties needed by the compiler. Thus, adding a new primitive means adding one entry to the table, and possibly an evaluation function. After that, the primitive instruction ise accepted by the scanner, scheduled appropriately, and printed out correctly. If the primitive has properties such as associativity and commutativity, optimizations which use these properties are able to use them. Similarly, if the primitive has an evaluation function, the various constant propagation transformations are able to use it.

Encapsulation is pervasive; data-structures are never accessed directly, but through functions or (more frequently) macros. This has enabled us to radically change the implementation of the various data-structures with no impact on the transformations.

Transformations are written to be stand-alone. They can be viewed as transformations on the DFG: they accept any possible DFG and transform it into some semantically equivalent DFG. A particular transformation may depend on a previous transformation in terms of the instruction-level parallelism it exposes, but not in terms of correctness.* The lack of required ordering enables the application of some subset of all optimizations, in an arbitrary order, speeding up the process of error isolation.

7. The design process and the simulation environment

Our design process and simulation environment have been *built around the architecture/compiler interaction*, leading to two paths as depicted in Figure 10:

- the *fast (development) path*, which is characterized by fast turn-around time but only instruction-set architecture performance measurements; and
- the *slow (performance evaluation) path*, which is characterized by longer turn-around time but performance measurements that take into account implementation aspects.

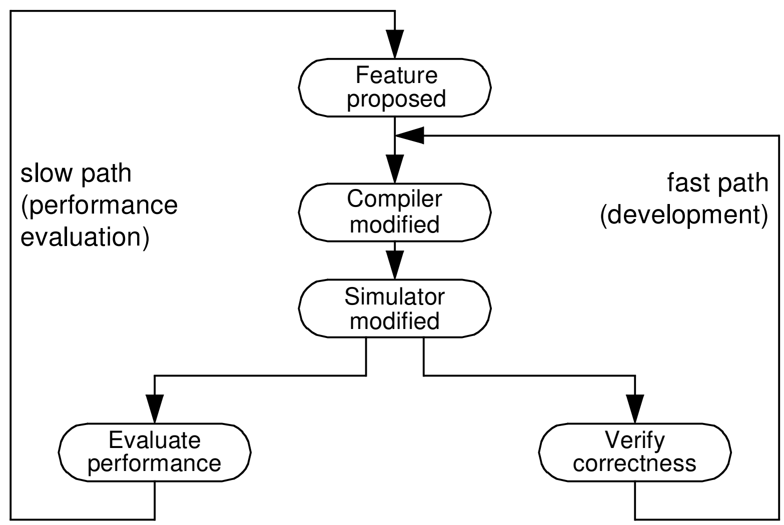

Figure 10: Iterative simulation/evaluation process

As their names imply, each path has a well-defined objective. The development path is used to introduce the changes required in the different parts of the environment, and their correctness is verified through a simulation step which provides only a preliminary performance estimate (without taking into account implementation issues such as memory effects, bypass interlocks, and so on). In contrast, the performance evaluation path assumes correct operation of the system, and focuses on providing an accurate estimate of the resulting performance, including the implementation aspects.

The two paths in the process above have been built into a simulation environment which comprises two phases, as follows (see Figure 11):

Preparation phase, in which assembly language code for the target architecture is translated into RS/6000 assembly code (on a file-by-file basis if the program consists of multiple files). The layout of data and procedure call conventions for the target architecture are the same as those used in RS/

* This is not strictly true; some transformations must be performed after register allocation.

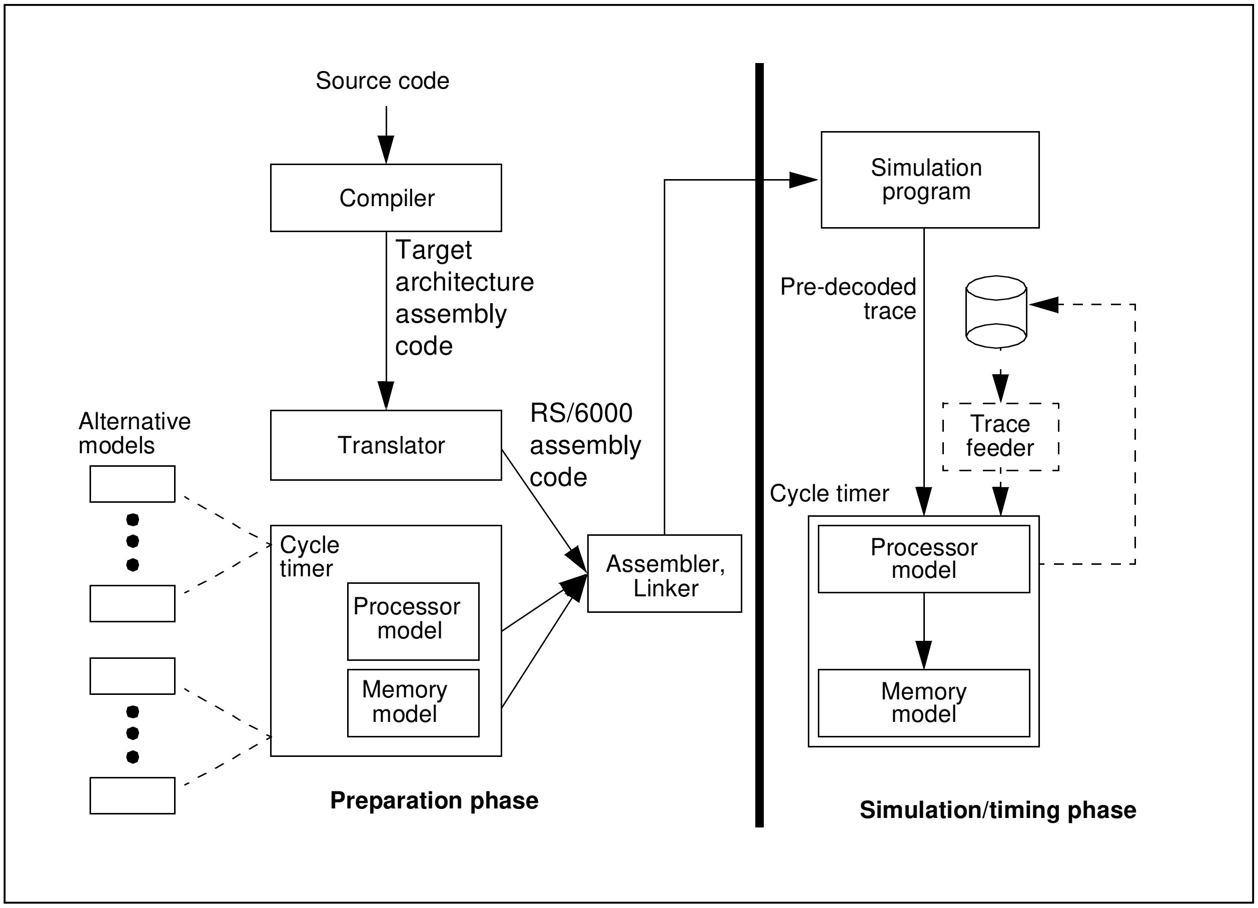

Figure 11: Overview of the simulation/timing environment

6000 and AIX. The resulting RS/6000 code is assembled and then linked with a cycle timer (processor and memory model).

Simulation/timing phase, in which the program for the target architecture is simulated, including the invocation of a cycle timer on a cycle-by-cycle basis.

This two-phase approach, which is common to many simulation/profiling tools [1], offers several special advantages in our case. For example, a given program for the target architecture can be used with alternative processor and memory models, providing varying degrees of detail in the performance data collected. On one extreme, no model is inserted whenever the desired objective is just the verification of correctness in the program, or just the collection of basic statistics regarding the instructions executed. On the other extreme, detailed processor and memory models are inserted to collect an accurate cycle count during execution of the program. The selection of a particular model is performed at link time, when the simulation executable is created.

Alternatively, the preparation phase allows using a trace-dump module instead of a processor model, in which case the execution trace of the program in the target architecture is stored on disk for later processing (this possibility is indicated by the dashed lines in Figure 11) or is used as input to other tools. Using this same approach, we have also envisioned the possibility of replacing the timer module with some form of debugging environment for the target architecture (similarly to what has being done with Shade [10]). These features are possible through specific program interfaces between processor model, memory model, and the emulated program.

Moreover, due to the same layout of data and procedure call conventions, the preparation phase allows *mixing assembly code from the VLIW architecture and RS/6000.* Since the translator generates RS/6000 assembly code on a file-by-file basis, it is possible to compile into code for the target architecture only a subset of the source files composing a program, and compile directly into RS/6000 assembly code the remaining files. In this way, the program resulting from the preparation phase only emulates and collects performance data for the part of the program which has been compiled for the target architecture, thus permitting focusing the development efforts only on critical parts of a program.

Mixing assembly code from the VLIW and RS/6000 architectures also allows invoking operating system resources in a straight-forward manner. The program in the VLIW architecture simply invokes the system function, and either a version in the target architecture or an RS/6000 version of that function is provided, depending on what was used at link time when the executable module was built. (Initially, we used the RS/6000 version of most of the system library functions.)

Components of the environment

More specifically, the components of the simulation environment are:

- An *optimizing compiler,* which generates assembly language code for the target architecture. The salient features of this compiler were discussed in the previous section.
- A *translator*, which maps the target assembly code into RS/6000 assembly code. The resulting code emulates the functionality of the target architecture for the specific program (as opposed to an interpreter using the target code as input). In addition, the RS/6000 code contains instrumentation to collect basic run-time data regarding the program in the target architecture as well as to generate pre-decoded execution traces.
- A *cycle timer*, invoked by the simulation code on a cycle-by-cycle basis, which processes the predecoded execution traces of the target architecture as they are generated. The cycle timer contains a processor model and a memory model, selected from a range of alternative models of varying level of detail and accuracy.

The translator

As stated above, the *translator* is used to convert assembly language code from the target architecture into the RS/6000 architecture (see Figure 12). Since the program in the target architecture may consist of instructions different from those available in the RS/6000 architecture, extra RS/6000 primitives must be added to simulate the effects of the new instructions. Moreover, since the target architecture may contain a larger register set than the RS/6000 set, registers of the target architecture are kept in memory, temporarily loaded into RS/6000 registers to perform specific operations, and results are stored back to the appropriate target registers in memory. This is basically the same scheme used in tools such as Shade [8].

Instruction in target architecture		**Translated RS/6000 code**

`add_shift r37,r4,r3,4`	→	`lwz r31,R4_off(r13)`
		`lwz r30,R3_off(r13)`
		`add r31,r31,r30`
		`sli r31,r31,4`
		`stw r31,R37_off(r13)`

Figure 12: Translation of assembly code from target architecture into RS/6000

Preliminary performance measurement capabilities are incorporated by the translator into the simulation executable through instrumentation for collecting data regarding the instructions executed. The measures correspond to statistics regarding the instruction-set architecture, and do not include details regarding an implementation of such an architecture. Thus, this mechanism allows fast turn-around time for experimenting with architectural features and compiler algorithms, without yet introducing the detailed description of a processor and memory implementation. (This is different from tools such as Shade, which only generate traces but do not collect any performance data because they are intended as drivers for separate tools modeling the processor.)

The instrumentation consists of counters which are placed at selected points in the code (points which represent specific entities in the program for the target architecture, such as for each VLIW). In addition, the translator generates a *file of descriptors,* each containing information which statically describes the operations and resources used by a program entity (i.e., for each VLIW). When the program finishes execution (a simulation run completes execution), the counters are saved in a file on disk; then, a postpass routine combines the information from the counters with the descriptors, thus computing performance data such as total number of times each VLIW was executed, average ILP achieved, utilization histograms for the different resources, and dynamic distribution of different opcode frequencies. The postpass routine is also used to generate profile-directed feedback information for the compiler.

The translator also places a call to the cycle timer for each program entity executed, passing as arguments the associated descriptor as well as an image of the registers specifying their contents just prior to the execution of the entity (i.e., a complete "trace record" which includes the status of the processor at that point); the descriptor contains a predecoded version of the program entity and predecoded information regarding the resources used by such entity. Only a pointer to these elements is actually used as argument in the call to the timer, so that there is no overhead in arranging data before the call.

The translator is driven by tables specifying the actual translation for each instruction defined in the target architecture. Adding a new instruction or modifying one already defined consists simply in changing the corresponding table entry. The type of instrumentation desired is a parameter to the translator, so that different levels of detail in the collection of data can be selected, with no runtime simulation overhead.

The cycle timer

The approach used in the cycle timer attempts to avoid some of the limitations of traditional timing models when processing traces of meaningful length (hundreds of millions of instructions), which either require very long execution time or rely on sampling the execution traces [26].* Two alternatives are provided for timing the behavior of a processor, both based on a cycle-by-cycle monitoring of the execution of a program:

1. A (traditional) trace-driven approach, in which an execution trace is first collected and then used as input to the timer, but using a trace which contains predecoded information regarding the resources required for the execution of each instruction.
2. An integrated simulation/timing approach, in which a program is simulated, a predecoded execution trace is generated, and the timing of the trace is performed on-the-fly.

In both cases the trace contains predecoded information, so that much of the overhead regarding processing the trace is eliminated. Moreover, in the case of the integrated approach, the overhead associated to accessing the trace is replaced by simpler procedure calls.

The efficiency of the timing environment is largely achieved through the use of the predecoded descriptors. A single read-only descriptor for each instruction is repeatedly used during simulation, so the size of the descriptor is not an important consideration. Thus, descriptors are designed to minimize the processing overhead of the cycle timer. In contrast, a conventional trace-driven timing environment typically

* In many instances sampling is regarded as inadequate to capture the actual performance expected from a processor. The environment described here may still need the use of sampling, but for much longer traces (programs running for many billion instructions).

must strive to minimize the size of the instruction and machine state information in the trace, at the expense of decoding overhead in the timer.

A descriptor is associated to an entire tree-instruction, so the overhead involved in invoking the cycle timer as well as in processing the descriptor is amortized over all the operations contained in the tree. Actually, a descriptor contains separate components for each path in the tree (*path descriptors)*, so that predecoded information is available for each path. As a result, during simulation the descriptor component associated to a taken path is used for timing the operations which are executed to completion, whereas the descriptors associated to non-taken paths can be used for timing conditional execution of operations for those cases where it might matter.

The *processor model* maintains the cycle count and other performance data, dealing with issues such as register dependencies and operation latencies for a given processor implementation. For memory operations, the processor model invokes the *memory model*, passing information such as the operation type and effective address of a memory reference. Since the processor model has access to the complete state of the processor, memory addresses can be regenerated so that complete checks on the address generation process are possible.

The processor and memory models each have a clearly defined interface, allowing a variety of models to be used interchangeably, with the models differing in both the system configuration they implement and in the degree of detail and accuracy involved. This versatility is further enhanced by ensuring that the interfaces provide all the information required by the most detailed or accurate model that may be needed, even if that information is not necessary for simpler models.

As is normally the case, the timing environment does not check for architecture restrictions; instead, it assumes that instructions have been executed properly, so they comply with the architecture. The verification of this assumption is performed by other support tools available in the environment.

We first developed simplified processor and memory models, which capture the basic features of an implementation. For example, a simple processor model detects and counts stall conditions such as register interlocks for long-latency operations (memory accesses, non-pipelined multi-cycle operations such as multiply and divide), bypass effects, conflicts in address generation, and conflicts in register targets. Similarly, a simple memory model describes a multilevel hierarchy (cache levels L0, L1, L2, and main storage), simulates only the corresponding directories, computes miss ratios at the different levels of the hierarchy, uses LRU replacement policies, assumes fixed latencies for transferring data between adjacent levels of the hierarchy, and delays the return of data to the processor based on such latencies; timing information is kept only for data moving through the memory hierarchy towards the processor but not for write-through data to percolate through all memory levels.

Timing information can be collected for specific routines rather than an entire program, by isolating those routines into separate source files and then generating timing support only for those files; this feature relies on the capability of the simulation environment to specify the compilation/instrumentation of selected files.

8. Some types of trade-offs

We now summarize some of the types of trade-offs which we have been performing with the environment described in this report; additional details on this subject are given in [9].

Operations and resources per VLIW

Important topics regarding a VLIW processor, and the trade-offs which are possible in terms of the hardware resources composing such a processor, are the effectiveness of the many functional units and the impact of the register set size. Consequently, it is interesting to explore the trade-offs among register set size and VLIW width (operations per long instruction). Typically, studies on VLIW architectures assume a fixed size register set and only investigate the effects of increasing the operations per VLIW [17] [22] [27].

These trade-offs are easily evaluated in our environment. Studying the impact of register set size and VLIW width requires invoking the compiler with a parameters file describing the features of the target architecture; no changes are required in the translator because this tool is capable of handling very large configurations (256 registers, unlimited number of operations per VLIW). We have used the environment to collect extensive data regarding these issues in the proposed VLIW architecture [9].

Collapsed dependent operations

Some processor architectures include in their instruction set the capability to perform certain pairs of dependent operations as a single instruction, so that the pair is executed in a single cycle [28]. A typical example is the sequence shift-add, in which a value is shifted by a small amount (say two or three bits), and the result of this operation is added to the contents of a register.*

We have explored the benefits of collapsed dependent operations in the context of our VLIW architecture and its compiler. The potential benefit of adding such pairs to the instruction set architecture is subject to the capability (or inability) of the compiler to hide the dependency among the corresponding operations as part of the execution of the entire program. Since the VLIW processor is characterized by having many functional units, the execution of an instruction pair as two separate instructions might not be detrimental as long the pair is not in the critical path of the program.

For this experiment, we first force the compiler to exclude the collapsed dependent operations and measure the resulting performance, and then we force the compiler to use them. The compiler includes a phase which searches for dependent pairs of instructions in successive VLIWs and merges them into one instruction, rescheduling the remaining operations as necessary. The translator recognizes the pairs and generates suitable simulation code. We have allowed the combination of any pair of arithmetic/logic operations, arithmetic or logic operations followed by a store, and a store followed by a load from the same location; we have also explored having only certain simple combinations (such as the shift-add pair indicated above).

The changes required for this type of experiments are carried out easily, given that the environment has been designed to support such modifications; the corresponding entries into the tables are added.

Data prefetch instructions

One important performance bottleneck in modern microprocessors arises from the long latency in accessing memory, in particular when the memory system is organized as a hierarchy of multiple levels of cache memory. Whenever a requested data item is not found in the first level of the memory hierarchy, there may be many cycles until data is brought in from another level of the hierarchy. One technique to cope with this problem consists of *non-blocking memory references*, in which case the processor may continue issuing and executing instructions beyond a cache miss, as long as the requested data is not yet needed for an operation [2].

Consequently, an important consideration is the placement of the memory references with respect to the actual usage of the data, so that potential long latencies in accessing data can be hidden. One way of helping this process consists on using *cache touch* instructions, which are hints to the hardware that a particular block of data might be needed soon in the execution of a program so it is advantageous to bring the corresponding data to the first level of memory, if possible. For example, the PowerPC architecture has a set of such instructions [31]. In spite of being defined in the architecture, there is still uncertainty about the bene-

* An out-of-order processor might find such dependent operations in the instruction stream even if they are separated by several instructions, and might be able to collapse them into a single operation. For these reasons, collapsing depending operations is considered an interesting proposition for existing architectures which do not have the pair in their instruction set.

fits of these instructions because they might generate contention for scarce resources which become busy for several cycles and which may be needed for other operations. On the other hand, it is widely accepted that such instructions can be used effectively in programs that are characterized by data accesses to structured data, as in technical/scientific computing.

We have investigated the trade-offs arising from data touch instructions in the context of our VLIW architecture/compiler combination. Given that an implementation of this architecture contains several load/store units capable of executing data touch instructions, the potential degradation due to contention for those resources with actual load/store instructions is not as severe as in the case of a processor with fewer resources. We have used different memory configurations, which are specified by simply providing parameters to the memory model. When necessary, we have defined smaller caches and larger problem sizes to force sufficient memory traffic beyond the first level of cache memory, but preserved long latency as if caches were large.

Some overall performance results

Additional data regarding instruction-level parallelism in programs, potential speedup with respect to a scalar processor, average number of useful instructions per VLIW, and so on, are reported in [9]. As an overall example, Table 1 lists the results for some programs in a configuration with 8-way branching and 16 functional units, of which at most eight can be load/store units and at most two can be floating-point units; the maximum size of a VLIW is 16 operations and 8-way branch. The data correspond to the ratios among the number of PowerPC instructions executed by the optimized native version of each program (as produced by the *xlc* compiler in AIX) with respect to the number of VLIWs executed by the VLIW program; these values correspond to a measure of path-length reduction achieved by the parallelizing compiler on the VLIW architecture.

Table 1: Path-length reduction in some programs

Program	Equivalent PowerPC instructions per VLIW
compress	5.4
eqntott	7.7
fgrep	7.5
lex	6.0
prof	4.7
sc	4.7
sed	5.8
uncompress	4.2
xlisp	3.9
yacc	4.4

9. Concluding remarks

We have described a novel processor architecture based on very-long instruction word principles, in which a program consists of a set of tree-instructions. The representation of programs in memory is implementation-independent, allowing their execution in processors with varying resources. The associated compiler uses state-of-the-art techniques to reach new levels of sustained instruction-level parallelism in branch-intensive and numeric-intensive programs, while also exhibiting much flexibility so that architecture/compiler/implementation trade-offs can be explored in multiple dimensions.

We have also described the approach used for the simulation/evaluation of the proposed VLIW architecture and its compiler. In practice, the approach and its associated environment are applicable to any type of processor architecture, but they are particularly well suited to modeling a VLIW-based processor due to lower simulation overhead. The simulation environment exhibits some unique features when compared to other tools with similar objectives, such as the ability to mix assembly code written for the target architecture with RS/6000 assembly code, and the ability to quickly provide basic performance data.

The environment is oriented to the evaluation of trade-offs in the architecture and its compiler, makes extensive use of table-driven techniques, and has been wholly designed for mutability. The performance of the tools has been sacrificed in those places where it was considered necessary in order to enhance the suitability for modifications. The environment provides fast turn-around time from compiler output to simulation results, allowing adequate support for testing new compiler algorithms and new architecture features. Thus, the environment is a good tool for the intended purposes, allowing performing extensive experiments to assess the potential benefits of the architecture/compiler combination, and the collection of extensive data supporting the evaluation of trade-offs in such a system.

In practice, the environment has allowed evaluating the performance of alternative architecture/compiler features over realistic work-loads, without having to store execution traces. Programs such as the SPEC92 benchmark suite, a set of AIX utilities, the LINPACK benchmark, and the Livermore loops benchmark have been simulated and timed in their entirety, for different processor configurations and different compiler algorithms. Simulation executables which do not invoke the cycle timer typically run only 10 to 15 times slower than the optimized native RS/6000 code for the same program; a processor model at the functional unit level and a memory model consisting of two levels of cache plus main memory slow the simulation executable by an additional factor of 75. These levels of performance in the simulator make possible carrying out complete experiments on a regular basis, without having to resort to simplifications to reduce their turn-around time.

References

[1] T. Conte, C. Gimarc, editors, *Fast simulation of computer architectures,* Kluwer Academic Publishers, 1995.

[2] J. Hennessy, D. Patterson, *Computer architecture - a quantitative approach,* 2nd. ed., Morgan Kaufmann Publishers, Inc., 1996.

[3] B. Rau, J. Fisher, "Instruction-level parallel processing: history, overview, and perspective," *The Journal of Supercomputing,* No. 7, pp.9-50, Kluwer Academic Publishers, 1993.

[4] MicroDesign Resources, *Microprocessor Report*, current year issues.

[5] J.A. Fisher, *The optimization of horizontal microcode within and beyond basic blocks,* Ph.D. Thesis, New York University, NY, 1979.

[6] J.A. Fisher, "Very long instruction word architectures and the ELI-52," in *Proceedings of 10th Annual International Symposium on Computer Architecture*, pp.140-150, 1983.

[7] L. Gwennap, "VLIW: The wave of the future?," *Microprocessor Report,* February 14, 1994.

[8] B. Cmelik, D. Keppel, "Shade: a fast instruction-set simulator for execution profiling," in *Fast simulation of computer architectures,* T. Conte, C. Gimarc, editors, pp. 5-46, 1995.

[9] "A tree-based VLIW processor: performance," Technical Report, IBM T.J. Watson Research Center, in preparation.

[10] J.A. Fisher, "Very long instruction word architectures and the ELI-52," in *Proceedings of 10th Annual International Symposium on Computer Architecture,* pp.140-150, 1983.

[11] R.P. Colwell, R.P. Nix, J.J. O'Donnell, D.B. Papworth and P.K. Rodman, "A VLIW architecture for a trace scheduling compiler," *IEEE Transactions on Computers,* Vol. C-37, No. 8, pp. 967-979, 1988.

[12] G.R. Beck, D.W.L. Yen and T.L. Anderson, "The Cydra 5 mini-supercomputer: architecture and implementation," *The Journal of Supercomputing,* Vol. 7, No. 1/2, pp.143-180, 1993.

[13] J.S. O'Donell, "Superscalar vs. VLIW," *Computer Architecture News*, Vol. 23, pp. 26-28, March 1995.

[14] R.L. Sites, A. Chernoff, M.B. Kerk, M.P. Marks, and S.G. Robinson, "Binary translation," *Communications of the ACM*, Vol. 36, pp. 69-81, February 1993.

[15] P. Koch, "Emulating the 68040 in the PowerPC Macintosh," in *Proceedings Microprocessor Forum*, October 1994.

[16] G. Silberman and K. Ebcioglu, "An architectural framework for supporting heterogeneous instruction-set architectures," *IEEE Computer*, Vol. 26, pp. 39-56, June 1993.

[17] T. Conte, S. Sathaye, "Dynamic rescheduling: a technique for object-code compatibility in VLIW architectures," in *Proceedings of 28th Annual International Symposium on Microarchitecture (MICRO-28),* 1995.

[18] B.R. Rau, "Dynamic scheduling techniques for VLIW processors," Technical Report HPL-93-52, Computer Research Center, Hewlett-Packard Company, June 1993.
[19] S. Melvin, M. Shebanow, and Y. Patt, "Hardware support for large atomic units in dynamically scheduled machines," in *Proceedings of 21st Annual International Symposium on Microarchitecture (MICRO-21)*, pp. 60-66, December 1988.
[20] M. Franklin and M. Smotherman, "A fill-unit approach to multiple-instruction issue," in *Proceedings of 27th Annual International Symposium on Microarchitecture (MICRO-27)*, pp. 162-171, December 1994.
[21] K. Ebcioglu, "Some design ideas for a VLIW architecture for sequential natured software," in *Parallel Processing (Proceedings of IFIP WG 10.3, Working Conference on Parallel Processing),* M. Cosnard et al. (editors), pp. 3-21, 1988.
[22] S-M. Moon, K. Ebcioglu, "An efficient resource-constrained global scheduling technique for superscalar and VLIW processors," *Proceedings of 25th Annual International Symposium on Microarchitecture (MICRO-25),* pp.25-71, December 1992.
[23] J.H. Moreno, "Object-code compatible representation of VLIW programs," Technical Report, IBM T.J. Watson Research Center, in preparation. (Patent Application, 1995, 1996.)
[24] J.H. Moreno et al., "Method and apparatus for reordering memory references in a superscalar or VLIW processor," Technical Report, IBM T.J. Watson Research Center, in preparation. (Patent Application, 1995).
[25] Pingali et al., "Dependence flow graphs: an algebraic approach to program dependencies," in *Proceeding of 18th ACM Symposium on Principles of Programming Languages,* pp. 67-78, 1991
[26] K. Menezes, "Sampling for cache and processor simulation," in *Fast simulation of computer architectures,* T. Conte, C. Gimarc, editors, pp. 171-203, 1995.
[27] P. Chang et al., "IMPACT: an architectural framework for multiple-instruction-issue processors", in *Proceedings 18th Annual International Symposium on Computer Architecture,* pp. 266-275, 1991.
[28] Hewlett-Packard Company, *PA-RISC 1.1 architecture and instruction set reference manual,* 2nd. edition, 1992.
[29] A.V. Aho et al., *Compilers: principles, techniques and tools,* Addison-Wesley, 1986.
[30] M.J. Wolfe, *Optimizing supercompilers for supercomputers,* MIT Press, 1989.
[31] IBM Corporation, *PowerPC Architecture,* 1st. edition, 1993.