Simulation/ evaluation environment for a VLIW processor architecture

by J. H. Moreno
M. Moudgill
K. Ebcioglu
E. Altman
C. B. Hall
R. Miranda
S.-K. Chen
A. Polyak

We describe the environment used for the simulation and evaluation of a processor architecture based on *very long instruction word (VLIW)* principles. In this architecture, a program consists of a set of tree instructions, each one containing multiple branches and operations which can be performed simultaneously. The simulation/evaluation environment comprises

- **An *optimizing compiler*, which generates tree instructions in a VLIW assembly language.**
- **A *translator* from VLIW assembly code into PowerPC® assembly code which emulates the functionality of the VLIW processor for the specific VLIW program. The emulating code also includes instrumentation for collecting execution counts of VLIWs, profiling information, and generation of predecoded execution traces.**
- **A *cycle timer*, invoked by the emulating code on a VLIW-by-VLIW basis, which processes VLIW execution traces as they are generated.**

The environment supports the evaluation of alternatives and trade-offs among the VLIW architecture, its compiler, and processor implementations. Emphasis has been placed on providing fast turnaround time for the development of compilation algorithms and an efficient compilation-to-simulation cycle which allows analysis of architecture/compiler trade-offs over complete execution runs of realistic workloads.

Introduction

The design of a new computer architecture and its associated compiler, 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.

0018-8646/97/$5.00 © 1997 IBM

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 varying assumptions regarding the features of software and hardware.

Because of 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 microarchitecture) includes trade-offs with respect to functional organization, logic design, and circuit design; some compiler functionality might be included in such trade-offs, though it seems that the compiler effects are usually taken into account after the microarchitecture has been developed. In contrast, the design of a new computer architecture frequently focuses 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. In general, the design process is an iterative one: Features are proposed, required changes are introduced to the design environment, and the effectiveness and performance of the new features are evaluated. Such a process is necessary because of the ever-tighter interaction among the different features. Consequently, the tools used should be efficient, allow experimentation with realistic workloads, be easy to reconfigure, and permit the evaluation of a variety of features. 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 turnaround time and the accuracy expected from the results.

- *Instruction-level parallelism*

An important focus of activity regarding computer architectures/microarchitectures in recent years has been the exploitation of instruction-level parallelism (ILP), that is, the ability to execute several operations simultaneously [1, 2]. 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 distinguish them from *scalar* processors, which dispatch at most one instruction per cycle. Moreover, when the set of operations issued for parallel execution is large (say larger than eight), the implementations are also known 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 adjacent set of operations that are executable simultaneously, and executes them in the order in which they appear in the program. In contrast, an out-of-order processor has hardware resources that determine the dependencies among successive operations as well as the order in which those operations are executed; such ordering may be different from the one in which instructions appear in the program. Most existing modern (high-performance) processor implementations belong to the out-of-order class.

The instructions executed simultaneously by an in-order issue superscalar processor are determined by the way in which instructions are placed in a program (that is, by the compiler or programmer), whereas an out-of-order processor discovers 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.

- *VLIW processors*

Aggressive in-order wide-issue execution leads to *very long instruction word (VLIW)* processors, wherein scheduling of instructions is done statically by a compiler that groups independent instructions executable in parallel, using optimization techniques such as software pipelining, loop unrolling, and scheduling code speculatively across basic blocks [2].

VLIW is not a recent concept; in fact, it originated as an extension of microcode-based techniques [3, 4], though the concept has changed drastically since its inception. As with all others, 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 run time 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 [5], "the objective of VLIW is to eliminate the complicated instruction scheduling and parallel dispatch that occur 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 a varying number of functional units [1].

• *Scope of the paper*

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

To support our research, we have created an environment for the simulation and evaluation of our VLIW architecture, which allows us to experiment with alternative features and to evaluate the benefits and complexities arising from such features. This environment provides *fast turnaround time for implementing new compilation algorithms,* and *fast turnaround 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.

In this paper, we describe the environment mentioned above. We focus mostly on features related to the instruction-set architecture and their relationship with compiler optimization algorithms. The relevance of the environment for experimental evaluation is described, emphasizing its ability to quickly introduce new compilation algorithms and rapidly incorporate and evaluate new architecture features. Quantitative results illustrate the mechanisms used to evaluate the abilities (or limitations) of the compiler to exploit the architectural features considered. In practice, the environment allows evaluation of alternative features over realistic workloads; programs such as the SPECint** benchmark suite and a set of AIX utilities can be routinely simulated in their entirety. Simulation at the instruction-set architecture level typically runs only seven to ten times slower than the optimized native PowerPC code for the same program. This level of performance in the simulator makes possible carrying out complete experiments on a regular basis, without having to resort to simplifications to reduce their turnaround time.

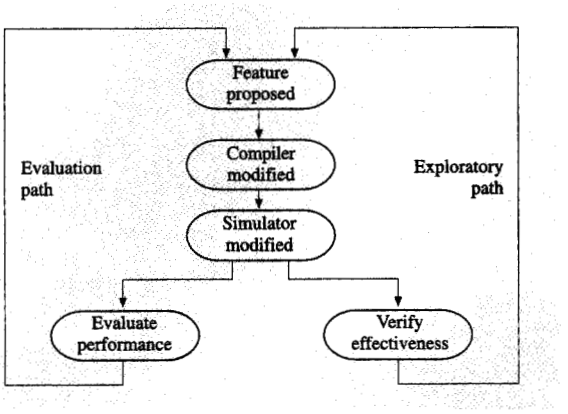

Figure 1

Iterative simulation/evaluation process.

The rest of the paper is organized as follows. We first describe the overall environment developed to support our research, and summarize how this environment differs from others previously reported in the literature. We then describe significant aspects of Chameleon, the optimizing compiler, and briefly summarize the basic properties of ForestaPC, our tree-based VLIW architecture. This is followed by a description of some examples of the architecture/compiler interactions which can be explored in our environment, illustrating them with quantitative results. We also provide data on the performance of the simulation/evaluation environment. We finalize with some observations regarding the compiler/architecture interactions, and the benefits of the environment.

The simulation and evaluation environment

Our simulation and evaluation environment has been *built around the architecture/compiler interaction*, leading to two paths, as depicted in **Figure 1**:

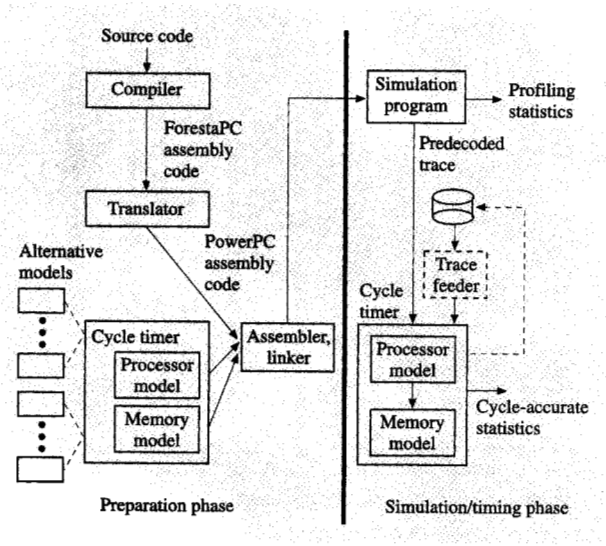

Figure 2

Overview of the simulation/timing environment.

- The *exploratory (fast) path*, which is characterized by fast turnaround time but only instruction-set architecture performance measurements.
- The *evaluation (slow) path*, which is characterized by longer turnaround time but performance measurements that take into account implementation aspects.

As their names imply, each path has a well-defined objective. The exploratory path is used to test new features by modifying the different components of the environment as necessary, and by simulating at the instruction-set architecture level (without taking into account implementation issues such as finite-size cache memories, interlocks, and so on). In contrast, the evaluation path focuses on providing accurate performance estimates, including the implementation aspects.

The two paths described above have been built into a simulation environment which comprises two phases (**Figure 2**):

- The *preparation phase*, in which ForestaPC (VLIW) assembly language code [6] is translated into PowerPC assembly code which emulates the behavior of the VLIW program (on a file-by-file basis if the program consists of multiple files).
- The *simulation/timing phase*, in which the VLIW program is simulated, including the collection of run-time profiling information and the invocation of a cycle timer on a cycle-by-cycle basis.

More specifically, the major components of the environment are the following:

1. An *optimizing compiler*, Chameleon, which generates assembly language code for the ForestaPC architecture. The salient features of this compiler are discussed in the next section.
2. A *translator*, which maps the VLIW assembly code into PowerPC assembly code. The resulting code emulates the functionality of the ForestaPC architecture for the specific program (as opposed to an interpreter using the target code as input). In addition, the resulting code contains instrumentation to collect run-time data regarding the VLIW program, and to generate predecoded execution traces.
3. A *cycle timer*, invoked by the simulation code on a cycle-by-cycle basis, which processes the predecoded execution traces of the VLIW 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 levels of detail and accuracy.

The two-phase approach illustrated in Figure 2, which is common to other simulation/profiling tools [7], offers several special advantages in our case. For example, the layout of data and procedure call conventions in the ForestaPC architecture are the same as those in PowerPC/AIX, so the preparation phase may mix assembly code from both architectures. Since the translator generates PowerPC assembly code on a file-by-file basis, it is possible to compile into ForestaPC assembly code only a subset of the source files composing a program, and compile the remaining files directly into PowerPC assembly code. In this way, the program resulting from the preparation phase emulates and collects performance data for only the part of the program that has been compiled for the ForestaPC architecture, thus permitting a focus on only the critical parts of a program. This helps to reduce the complexity and execution time of the simulation and evaluation tasks.

Moreover, the resulting PowerPC code is assembled and then linked with the cycle timer, which includes a processor and a memory model. Consequently, a given program for the ForestaPC architecture can be used with alternative processor and memory models, providing varying degrees of detail in the performance data collected. At 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. At 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 may create a simulation module for generating and saving the execution trace of the ForestaPC program instead of consuming the trace on the fly; this possibility is indicated by the dashed lines in Figure 2. Using this same approach, we have also envisioned the possibility of replacing the cycle timer module with some form of debugging environment for the ForestaPC architecture (resembling what has been done with the tool Shade [8]). These features are possible through specific programming interfaces among the processor model, the memory model, and the emulated program.

Mixing assembly code from the ForestaPC and PowerPC architectures also allows the invocation of operating system resources in a straightforward manner. The program in ForestaPC assembly code simply invokes the system function, and either a VLIW version or a PowerPC version of that function is provided at link time (initially, we used the PowerPC version for most of the system library functions).

- *The translator*

As stated above, the translator is used to convert ForestaPC assembly language code into PowerPC code (see **Figure 3**). Since the program in the ForestaPC architecture may contain instructions different from those available in the PowerPC architecture, extra PowerPC primitives must be added to simulate the effects of the new instructions. Moreover, since the VLIW architecture may contain a larger register set than the PowerPC set, registers of the VLIW architecture are kept in memory and temporarily loaded into PowerPC registers to perform specific operations, and results are stored in the appropriate target registers in memory. This is basically the same scheme used in tools such as Shade [8].

Preliminary performance measurement capabilities are incorporated by the translator through instrumentation for collecting data regarding the VLIWs executed. The measures correspond to statistics on the instruction-set architecture, and do not include details regarding an implementation of such an architecture. Thus, this mechanism allows fast turnaround time for experimenting with architectural features and compiler algorithms, without yet introducing the detailed description of a processor and memory implementation.

The instrumentation consists of counters which are placed at selected points in the code (representing each exit point from a VLIW). In addition, the translator generates a *file of descriptors*, each containing predecoded information which describes the operations and resources used by a VLIW. When the program finishes execution (a simulation run completes execution), the counters are saved in a file on disk; then, a separate tool combines the information from the counters with the descriptors, thus

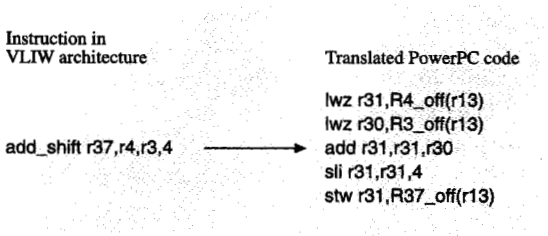

Figure 3

Translation of VLIW assembly code into PowerPC code.

computing performance data such as total number of times each VLIW was executed, average ILP achieved, utilization histograms for the different resources, and dynamic frequency distribution of the different primitive instructions. The tool also generates profile-directed feedback information for the compiler, though we do not currently exploit that capability.

The translator also places a call to the cycle timer for each VLIW 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 VLIW (i.e., a complete "trace record" which includes the status of the processor at that point). Only a pointer to these elements is actually used as argument in the call to the cycle timer, so there is no overhead in arranging data before the call.

The translator is driven by tables specifying the actual translation for each primitive instruction defined in the ForestaPC 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 different levels of detail in the collection of data can be selected at translation time.

- *The cycle timer*

The approach used in the cycle timer attempts to avoid some of the limitations of traditional timing models in processing traces of meaningful length (hundreds of millions of instructions), which either require a very long execution time or rely on sampling of the execution traces[1] [9]. 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

[1] The environment described here may still require sampling, but for much longer traces (programs running for many billions of instructions).

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 in processing the trace is eliminated. Moreover, in the case of the integrated approach, the overhead associated with accessing the trace is replaced by simple 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 VLIW is repeatedly used during simulation, so the size of the descriptor is not an important consideration. The overhead involved in invoking the cycle timer as well as in processing the descriptor is amortized over all of the operations contained in the tree. Thus, descriptors are designed to minimize the processing overhead of the cycle timer. In contrast, a conventional trace-driven timing environment typically 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.

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*, complete checks can be performed on any processor functionality (i.e., fast memory address generation, prefetching, etc.).

The processor and memory models have a clearly defined interface, allowing a variety of models to be used interchangeably, with the models differing both in 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 of the information required by the most detailed or accurate model that may be needed, even if that information is not necessary for simpler models.

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, the descriptor component associated with a taken path is used for timing the operations which are executed to completion, whereas the descriptors associated with nontaken paths are used for timing conditional execution of operations for those cases where it might matter.

As is normally the case, the timing environment does not check for architecture restrictions; instead, it assumes that instructions have been compiled properly, so that 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, nonpipelined multicycle operations such as multiply and divide), register-bypassing 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 on the basis of such latencies; timing information is kept only for data moving through the memory hierarchy toward the processor, 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 for only those files; this feature relies on the capability of the simulation environment to specify the compilation/instrumentation of selected files.

- *Relationship with other simulation/evaluation tools*

A good summary of the state of the art in simulation of processor architectures is given in [8], which contains an extensive list of related tools as well as a description of Shade, perhaps one of the most complete and effective simulation and tracing tools currently available. Since attempting to illustrate the similarities and differences among other tools and ours is beyond the scope of this paper, we compare relevant features of our environment with those provided by Shade, and summarize the unique properties of our environment. Further comparisons with other tools can be inferred from the summary given in [8].

As stated in [8], Shade is characterized by the following features:

- Library functions invoked by analyzer programs are collected; analyzer programs typically use Shade to collect raw trace data, and then summarize/process the data to provide specific metrics.
- Executable code for a target machine is dynamically cross-compiled into executable code that runs directly on a host machine.

- Host code is cached for reuse so that the cost of cross-compiling can be amortized.
- Simulation and tracing code are integrated so that the host code saves trace information directly as it runs.
- The analyzer is given control over what is traced, and the tracing strategy can be varied dynamically.

While our environment shares most of the objectives and capabilities of Shade, it also has some very distinctive features. In particular, we rely on static translation prior to simulation instead of dynamic translation, thereby reducing run-time overhead at the cost of storage space. Moreover, our environment relies on allowing mixing assembly code for the target architecture with assembly code for the host architecture, thereby reducing the complexity of the tasks; such a capability is not available in Shade because translation is performed from the executable code for the target architecture of the entire program.

Both Shade and our environment give the processor model (analyzer in Shade's terms) control over the tracing strategy, but the approach used is different. Shade generates the trace records dynamically, according to the model's specifications. In contrast, since we rely on static translation, the processor model receives a pointer to the complete trace record, with no run-time overhead, and the model uses as much information from this record as necessary.

In addition, our environment is based on the availability of a suitable optimizing compiler, so that architecture/compiler interactions are explored in many dimensions. In contrast, Shade operates on executable code produced independently; it is not intended to explore such types of interactions. In this context, it would be more appropriate to compare our approach with tools that have similar objectives, such as the environment supporting IMPACT [10], but that environment is not as general as Shade or the one described here.

The Chameleon compiler

Chameleon, our research compiler (see **Figure 4**), has been designed to support research into instruction-level parallelism, and to evaluate the benefits of various architectural modifications when exploited through appropriate compiler optimizations. Chameleon was designed to target VLIW architectures that execute tree instructions; it is extensively parameterized so that it can target processors with different features (such as issue width, number of functional units, instruction latencies, and register set). Supporting architectural explorations and implementing aggressive optimizations geared toward several different targets implies a compiler in a constant state of change. Consequently, the compiler has been designed with support for the addition of code, verification

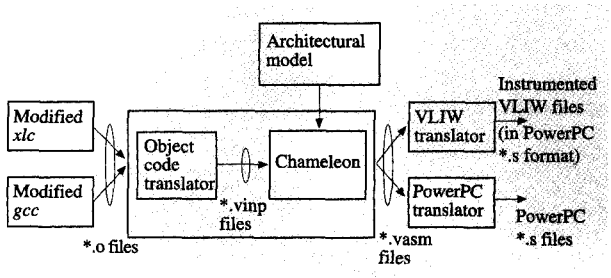

Figure 4

The Chameleon compiler environment.

of the modified compiler, and rapid isolation of problems (such mutability has given it its name).

The input to Chameleon is object code (*.o files) produced either by a modified version of *xlc*, the standard RS/6000* C compiler [11, 12], or a modified version of *gcc*, the GNU C compiler. The object files are processed by an "object-code translator" that generates an assembly-like sequential representation (*.vinp files). The output from Chameleon is a ForestaPC program (tree instructions) in an assembly language form (*.vasm files), which is either instrumented and translated into PowerPC assembly code (*.s files) that emulates the target ForestaPC processor, or is directly translated into PowerPC assembly code.[2]

The modifications to *xlc* and *gcc* fall into two categories: First, we have turned off phases such as scheduling and loop transformations that would tend to obscure the original code sequence; second, we have added capabilities to convey information such as alias classes, spill locations, and registers live at function calls and function return points. The results reported in this paper have all been obtained using *xlc*. Among other benefits, the modified *xlc* allowed us to take advantage of an existing production compiler front end.

In contrast to many other compilers, our target architecture is not fixed. Since one use of the compiler is investigating trade-offs among features such as the size of the register set and the number of execution units, 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 it possible to determine the performance of compiled code with and without an additional feature.

In summary, the compiler is characterized by its ability to generate optimized code to properly exploit a VLIW

[2] Such code is suitable for a wide-issue implementation of the PowerPC architecture.

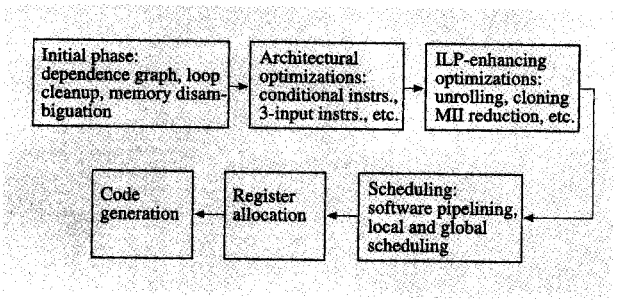

Figure 5

Phases in Chameleon.

architecture; its ability to support multiple processor configurations, so that the number and balance of various resources can be studied; and its modifiability to allow exploration of novel compiler optimizations and architecture-related ideas.

- *Optimizations*

Chameleon has a fairly aggressive suite of optimizations which can be grouped into the following categories (see **Figure 5**):

- *Traditional* Optimizations including constant propagation, loop-invariant code motion, dead-code elimination, and subexpression elimination; these are applied throughout the compilation process. Moreover, since Chameleon uses the output from *xlc*, it takes advantage of *xlc*'s excellent set of traditional optimizations.
- *ILP-increasing* Optimizations designed for increasing the instruction-level parallelism, so that the scheduler can pack instructions tightly. These include various loop transformations, such as unrolling, rewriting loops with commutative/associative operations (reductions), rewriting cyclic dependences to reduce initiation intervals, and memory disambiguation.
- *Architectural* Optimizations designed to exploit various architectural extensions. For instance, there is a phase that uses conditional move/store operations to convert if-then-else structures to straight-line code.

Optimizations not in Chameleon but found in other research compilers include profile-directed feedback and interprocedural analysis. Chameleon has the ability to use profiling information, but it normally uses synthetic branch probabilities produced by a variant of the Ball–Larus heuristics [13]. We do not exploit interprocedural analysis; even the interprocedural phase of *xlc* has been disabled.

The scheduler used by Chameleon is an enhanced version of selective scheduling [14]. The original algorithm has been considerably modified; the changes include examining all instructions in a loop for scheduling (instead of only instructions within a fixed "window"), using heuristics for selecting the slot in which to schedule an instruction (instead of a "greedy" schedule), and sensitivity to register pressure (not scheduling an instruction if that might cause a register spill).

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

Much 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 the following:

- *Traditional address disambiguation* We use aggressive versions of traditional and VLIW techniques [15–17], computing 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 [18].
- *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. Our loop-cloning implementation is a more aggressive version of the technique described in [19].
- *Reordering memory operations* A software-based *coherence test* is used for reordering load instructions relative to store instructions, in spite of the possibility of conflicts due to memory references which cannot be disambiguated [20]. Whenever a load instruction is moved earlier than a sequentially preceding ambiguous store instruction, a coherence test is inserted at the original position of the load instruction in the sequential instruction stream. The coherence test consists of two instructions: a load instruction from the same memory location, followed by a *trap if not equal* instruction which compares the value just loaded with the value loaded out of order. If the values are identical, the value loaded out of order and all other values derived from it are correct, and execution can proceed normally. On the other hand, if the value just loaded is different from the

value loaded out of order (which implies that the corresponding memory location has been modified after being read), the value loaded out of order, as well as all other values derived from it, is incorrect and must be recomputed. A trap handler is invoked, activating recovery code generated by the compiler, which re-executes the out-of-order load instruction as well as those instructions that depend on it and were executed before the trap was generated. For these purposes, the instructions executed between the out-of-order load instruction and the coherence test must not destroy the operands of the instructions that are re-executed in the recovery code.

Loop rewriting

Loops, especially in numerically intensive code, contain many potentially parallel instructions. We use several loop-rewriting optimizations [18] to transform loops and increase the amount of ILP available. These transformations include the following:

- *Unrolling loops* As implemented, software pipelining can start at most one iteration of a loop at a time. Unrolling small loops allows a scheduler to be presented with 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; the partial results are then combined to obtain the actual value. On the basis of 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.

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 reordering of memory operations already mentioned, the supported features include

- *Nontrapping/speculative load instructions*, which enable speculative issuing of load instructions, 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 permit conversion of if-then-else structures into a mixture of speculative operations and select instructions, thereby simplifying 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.

- *Implementation*

Chameleon is table-driven, so that adding a new instruction and/or a new register class requires localized changes. Its intermediate form, the dependence flow graph (DFG) [21], provides an integrated data/control flow information well suited for incorporating advanced optimizations; such optimizations are, with few exceptions, independent of one another and permutable.

An exhaustive list of the benefits of using the DFG representation is beyond the scope of this paper. Among other benefits, DFGs combine control-flow and dependence information into a single coherent form. They are fully executable representations, and 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.

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 of 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).

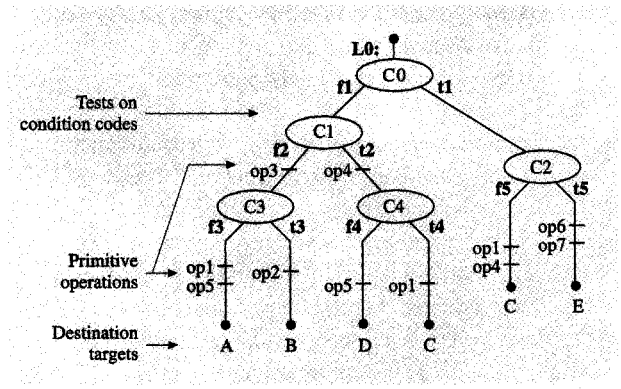

Figure 6

Tree instruction.

Chameleon is extensively parameterized; everything from the processor resource model to the instruction set is defined through tables, which are usually modifiable at run time. For instance, primitive operations and their properties are in a table. Thus, adding a new primitive means adding one entry to the table, and possibly an evaluation function. After that, the primitive instruction is 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.[3] 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.

Chameleon has extensive debugging support for reducing the time to isolate and fix programming errors, including built-in data-structure consistency and checking for memory bounds/validity. A nonoptimized version of the compiler, with debugging enabled, can spend up to two thirds of its execution time in asserts and data-structure integrity validation. This property has proven very effective; 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.

Compilation time

The area in which our compiler differs most from others is compilation time, though this is the result of conscious decisions. Whenever there was a trade-off among compilation time and other properties such as robustness, maintainability, extensibility, performance, or programmer productivity, we chose against compilation time. For instance, we frequently use global $O(n^2)$ algorithms over entire functions. In addition, while DFGs are ideal for implementing optimizations, they are more memory-intensive than other intermediate forms.

Basic features of the VLIW architecture

In ForestaPC, our VLIW 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 6**) [6]. 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. The structure of these tree instructions is an extension to those described in [14, 22].

On the basis of 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 are 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 produce no effect on the state of the processor.

Primitive instructions in a tree are subject to *sequential semantics for each path*, as if each primitive instruction were executed in the order in which it appears in the tree path (a tree path starts from the root of the tree and ends in a destination target). As a result, a primitive instruction cannot use a processor resource which is set by a previous instruction in the same tree path. This property guarantees binary compatibility among different implementations of the ForestaPC architecture with varying degrees of parallel execution capabilities, because

[3] This is not strictly true; some transformations must be performed after register allocation.

large trees can be decomposed into subtrees which are executed in different cycles [23, 24].

More specifically, a ForestaPC processor fetches tree instructions from main storage for execution. If the size of a tree instruction exceeds the resources in the processor (such as number of branch operations or number of fixed- or floating-point operations), the tree instruction is dynamically decomposed (*pruned*) to fit the resources available in the processor. The resulting subtrees are executed in successive cycles, unless the taken path is completely contained within the first subtree.

Additional details regarding the ForestaPC architecture are described in [6, 23–25].

Some examples of compiler/architecture interactions

The range of architecture/compiler interactions which can be explored in our environment is quite broad. Since we cannot describe the entire range in detail, we list below some examples of architectural features which have been considered and for which compiler algorithms have been developed:

- Number and type of operations per VLIW.
- Size of the register set.
- Latencies of operations, including memory operations.
- Availability/unavailability of specific instructions.
- Three-input instructions.
- Conditional move and conditional store instructions.
- Record form of instructions.
- Length of displacement and immediate fields.
- Static reordering of ambiguous memory references, with run-time verification of incorrect execution.
- Cache prefetch instructions.

We now illustrate the capabilities of the environment through two specific examples—the exploration of issue width with larger register set, and the incorporation of three-input operations in the architecture.

- *Issue width and register set size*

Typically, studies on VLIW architectures assume a fixed-size register set and investigate the effects of increasing the operations per VLIW [10, 14, 26]. In addition to performing such studies, we have explored the availability of instruction-level parallelism assuming a larger register set for wider-issue implementations. Such a trade-off is easily evaluated in our environment. The compiler is simply invoked with a parameters file describing the features of the target architecture. No changes are required in the translator, because that tool is capable of handling very large configurations (1024 registers, unlimited number of operations per VLIW).

Table 1 Size of register set in primitive operations.

Register class	*Issue width*		
	8	*12*	*16*
General-purpose	64	96	128
Floating-point	64	96	128
Condition	16	24	32

Table 2 Size of latencies in primitive operations.

Operation	*Latency*
Integer	1
Floating-point	3
Load	1
Integer divide	10
Integer multiply	3

Table 3 VLIW instruction ratio with respect to sequential code.

Benchmark	*Issue width*		
	8	*12*	*16*
SPECint92			
compress	4.14	5.14	5.79
eqntott	4.79	5.00	8.02
espresso	2.58	2.85	3.11
gcc	2.36	2.76	2.88
li	3.25	3.58	3.69
SPECint95			
m88ksim	2.70	3.02	3.04
go	2.23	2.45	2.51

As an example, **Tables 1** and **2** respectively list the size of the register sets and the operation latencies considered for one experiment, in which we evaluate the instruction-level parallelism found for three different processors capable of issuing *any* eight, twelve, or sixteen instructions per VLIW. The programs used in this experiment are taken from the SPECint92** and SPECint95** suites.

Table 3 lists the instruction-level parallelism found for the three processor configurations considered. This figure indicates the ratio between the number of instructions executed by a processor running PowerPC code (with ILP = 1) and the number of tree instructions executed by the ForestaPC processor, for the different issue widths. The PowerPC instruction counts used for these ratios are obtained by compiling the programs with *xlc* at optimization level O2.

Table 4 Relative ILP gain from three-input operations in 16/8/4/2/16 processor.

Benchmark	*Base ILP*	*3-input ILP*	*Gain (%)*
compress	4.41	5.38	18.0
eqntott	7.88	7.91	0.4
espresso	2.78	2.90	4.1
gcc	2.65	2.68	1.1
li	3.48	3.53	1.4
m88ksim	2.80	2.84	1.4
go	2.08	2.39	13.0

Note that we compute instruction-level parallelism differently from many other results reported in the literature. Usually, the results reported are obtained by using the same compiler for both the parallel implementation and the sequential implementation. In contrast, the instruction counts for the sequential implementations are obtained using the best compiler available for the PowerPC architecture. This is motivated by our original research goal, namely measuring the potential improvement in instruction-level parallelism in a PowerPC-based VLIW processor over existing PowerPC implementations. Moreover, we believe this is the proper way to compute ILP.

- *Three-input operations*

Earlier work has shown that it is possible to build hardware that can combine two arithmetic-logical operations into a single one, and analysis of execution traces has indicated that there are opportunities for taking advantage of such combinations [27, 28]. For example, an add&shift instruction is a three-input operation that performs the addition of two operands followed by shifting the intermediate result a number of positions specified by a third operand; that is, r5=add&shift r3,r4,r7 is equivalent to rx=add r3,r4 followed by r5=shift rx,r7. In fact, contemporary architectures such as Hewlett-Packard's PA-RISC** [29] have some capabilities of this type.

Note that the potential benefit of adding three-input instructions to an architecture is subject to the capability (or inability) of the associated compiler to hide the dependency among the corresponding operations as part of the execution of the entire program. Since a 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 as the pair is not in the critical path of the program (neglecting penalties arising from having larger code size).

We have used our environment to explore the benefits of including combined operations in the VLIW architecture. Initially, we considered the following classes of three-input operations (a total of 67 additional instructions):

- A: Any combination of add/subtract with add/subtract.
- S: Any combination of add/subtract with shift, or shift with add/subtract.[4]
- L: Any combination of logical with logical operations.

Moreover, for determining an upper bound on the potential performance achievable, we also allowed "recording" forms of each of these combinations (i.e., setting a condition register in addition to the result), as well as specifying an immediate value for one of the operands. Because of encoding constraints, these combinations require the use of a doubleword for their representation in memory.

We first added these three-input operations to the compiler and to the simulator. In Chameleon, this required adding an entry for each instruction in the opcode table, and an evaluation function. The translator was modified to recognize the new operations, decompose them into their two-input components, and emit suitable PowerPC assembly code emulating the new instructions.

The next step was adding the necessary optimizations to Chameleon. We made three changes for exploiting the availability of three-input operations:

- Added a new phase that combines two operations into a three-input operation when the two operations are in the same basic block.
- Modified the scheduling heuristics to properly handle the single-cycle latency of a three-input operation, thereby combining instructions when such an action would produce a better schedule.
- Altered the final peephole compaction phase so that combining adjacent VLIWs also recognizes and exploits the three-input operations.

As an example, **Table 4** depicts the relative gain in instruction-level parallelism arising from this interaction among compiler and architecture, for the case of a processor capable of issuing up to 16 operations per VLIW but restricted to eight memory operations, four-way branch, and two floating-point operations, and whose register set is 64/64/16 registers. As listed in the table, some programs exhibit significant gains, whereas others reflect little variation.

For the same example, **Table 5** illustrates the distribution of the most common three-input operations for the benchmarks *compress, espresso*, and *go*, which are the only ones that exhibit significant gain from the availability of the new instructions (see Table 4). In Table 5, the static instruction count represents the number of occurrences of those specific instruction combinations which are present in the VLIW program, whereas the

[4] This class does not include arithmetic shift instructions.

dynamic ratio corresponds to the ratio between the dynamic count of such specific instruction combinations and the total number of operations in the entire program (specified in either taken or nontaken paths of the tree instructions).

Finally, **Table 6** depicts the effects of the three-input instructions grouped according to their classes, for the same three benchmarks as in Table 5. In the case of *compress* and *go*, the most relevant group is A + S; the L group does not have any effect. In contrast, in the case of *espresso*, each class contributes partially to the overall gain. However, this example indicates that the gain arising from the availability of two classes is not necessarily the same as the sum of the individual gains; the compiler is able to schedule instructions in such a way as to partially compensate for a missing class.

• *Simulation performance*

As was stated earlier, a ForestaPC program simulated at the instruction-set architecture level executes seven to ten times slower than the optimized native version of the same program, whereas simulation including a processor and memory model adds an extra slowdown factor of the order of 100. This is illustrated in **Table 7** for some workloads: *lex* and *yacc*, two AIX utilities, and *compress* and *alvinn* from the SPEC92** suite. In these experiments, the processor model counts stall cycles due to implementation issues such as interlocks required by long-latency operations, long bypass paths, carry-out from fast address addition, and conflicts in accessing ports to the register file. In turn, the memory model implements a three-level cache hierarchy, with fixed latencies for transferring data between adjacent levels of the hierarchy toward the processor, and main memory.

Table 7 indicates the number of tree instructions executed in each program, the time spent in user code for the native version of the program (the time spent in system code is not included because it is negligible), the time spent in user code in the exploratory path and in the evaluation path, and the ratios of these values with respect to native execution. The entry for benchmark *compress* corresponds to execution on the official short input, whereas *alvinn* corresponds to the official reference input (1.3 billion VLIWs executed).

As could be expected, the slowdown factor varies across different programs, in particular for simulation with a processor model (the evaluation path). The most important factor contributing to the differences is the frequency of memory operations in the programs. Since memory operations imply invocations of the memory model, the slowdown factor is larger for programs with higher memory traffic; for example, about 38% of the operations performed in *compress* are memory operations, in contrast to about 62% in benchmark *alvinn*.

Table 5 Distribution of three-input operations in benchmarks *compress*, *espresso*, and *go*.

Benchmark	*Operation*	*Static instruction count*	*Dynamic ratio (%)*
compress	slw,add	45	10.5
	add,subf	23	8.1
	add,add	56	3.0
	srw,add	4	<0.01
	all others	1	0.0
espresso	add,add	1145	3.5
	and,nor	86	1.7
	slw,add	1496	1.7
	and,and	38	0.5
	all others	453	1.2
go	slw,add	45	10.5
	add,subf	23	8.1
	add,add	56	3.0
	srw,add	4	<0.01
	all others	1	0.0

Table 6 Relative ILP gain from classes of three-input operations.

Instruction classes	*Benchmarks (%)*		
	compress	*espresso*	*go*
A + S + L	18.0	4.1	13.0
A + S	18.0	3.2	13.0
A	11.2	2.2	0.5
A + L	11.2	2.8	0.5
S	7.3	1.3	12.1
S + L	7.3	2.2	12.1
L	0.0	0.8	0.0

Concluding remarks

We have described the simulation and evaluation environment for a VLIW architecture and its compiler, and have illustrated some examples of the application of the tools. In practice, the approach used is applicable to any type of processor architecture, but it is particularly well suited to modeling a VLIW-based processor because of lower simulation overhead.

The environment is oriented toward the evaluation of trade-offs in a VLIW architecture and its compiler, makes extensive use of table-driven techniques, and has been wholly designed for mutability. The environment provides fast turnaround time for introducing new compilation algorithms, and fast turnaround time from compiler output to simulation results, allowing adequate support for testing of compiler algorithms and architecture features. Thus,

Table 7 User execution time.

Benchmark	*VLIWs* $\times 10^6$	*Native* (s)	*Exploratory path*		*Evaluation path*	
			(s)	*Ratio*	(s)	*Ratio*
compress	1.54	0.60	3.40	5.67	309	515
yacc	16.56	0.97	7.27	7.49	762	786
lex	2.33	2.04	14.97	7.34	1337	655
alvinn	1300	35.80	369	10.32	53490	1507

the environment allows the performing of extensive experiments to assess the potential benefits of the VLIW architecture/compiler combination, and the collection of extensive data supporting the evaluation of trade-offs in such a system.

In practice, the environment has permitted routine evaluation of alternative architecture/compiler features over realistic workloads. Programs from the SPEC92 and SPEC95** benchmark suites, 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 seven to ten times slower than the optimized native PowerPC 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 the order of 100. These levels of performance in the simulator make possible complete experiments on a regular basis, without having to resort to simplifications to reduce their turnaround time.

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

- Highly modular organization.
- An optimizing compiler integrated with the development of the processor architecture.
- Fast turnaround time for introducing compiling optimization algorithms.
- Fast turnaround time from compilation output to simulation results.
- 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 ability to mix assembly code written for the target architecture with assembly code for the host architecture.
- The use of static translation and predecoding to reduce run-time overhead.
- The capability to obtain different levels of accuracy in the performance measures, with more accurate results requiring longer execution time.
- Applicability to any type of processor architecture or microarchitecture, particularly for modeling wide-issue processors such as VLIW because of lower simulation overhead.

The techniques and the environment developed as part of this research are applicable more generally than just for the case of in-order issue processors as described in this paper, though some of the benefits are clearer in such a context. In any case, we envision further work on the development of similar environments for the simulation and evaluation of aggressive in-order and out-of-order processors, or for other new processor architectures; the compiler can be used to further explore the potential and limitations of instruction-level parallelism, either in unconstrained conditions or constrained by specific architecture features, and for other languages and programming environments.

Acknowledgments

We thank Norman Cohen, Richard Goldberg, Peter Oden, Vladimir Kotlyar, Induprakash Kodukula, and Balaram Sinharoy for their contributions to Chameleon.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Standard Performance Evaluation Corporation or Hewlett-Packard Company.

References

1. J. Hennessy and D. Patterson, *Computer Architecture—A Quantitative Approach*, 2nd edition, Morgan Kaufmann Publishers, Inc., San Francisco, 1996.
2. B. Rau and J. Fisher, "Instruction-Level Parallel Processing: History, Overview, and Perspective," *J. Supercomputing* **7**, No. 1/2, 9–50 (1993).
3. J. A. Fisher, *The Optimization of Horizontal Microcode Within and Beyond Basic Blocks*, Ph.D. Thesis, New York University, New York, 1979.
4. J. A. Fisher, "Very Long Instruction Word Architectures and the ELI-52," *Proceedings of the 10th Annual*

International Symposium on Computer Architecture, 1983, pp. 140–150.
5. L. Gwennap, "VLIW: The Wave of the Future?," *Microprocessor Report*, February 14, 1994, pp. 18–21.
6. J. H. Moreno, K. Ebcioglu, M. Moudgill, and D. Luick, "ForestaPC (Scalable VLIW) User Instruction Set Architecture," *Research Report RC-20733*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1996.
7. *Fast Simulation of Computer Architectures*, T. Conte and C. Gimarc, Eds., Kluwer Academic Publishers, Boston, 1995.
8. B. Cmelik and D. Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling," *Fast Simulation of Computer Architectures*, T. Conte and C. Gimarc, Eds., 1995, pp. 5–46.
9. K. Menezes, "Sampling for Cache and Processor Simulation," *Fast Simulation of Computer Architectures*, T. Conte and C. Gimarc, Eds., 1995, pp. 171–203.
10. P. Chang, S. Mahlke, W. Chen, N. Water, and W. Hwu, "IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors," *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991, pp. 266–275.
11. IBM Corporation, *IBM CSet++ for AIX*, Order No. 5C09-1968-01, 1995.
12. M. Auslander and M. Hopkins, "An Overview of the PL.8 Compiler," *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, June 1982, pp. 22–31.
13. T. Ball and J. Larus, "Branch Prediction for Free," *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation*, June 1993, pp. 300–313.
14. S.-M. Moon and K. Ebcioglu, "An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors," *Proceedings of the 25th Annual International Symposium on Microarchitecture* (*MICRO-25*), December 1992, pp. 25–71.
15. A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley Publishing Co., Reading, MA, 1986.
16. J. Ellis, *Bulldog: A Compiler for VLIW Architectures*, MIT Press, Cambridge, MA, 1986.
17. S. M. Moon and K. Ebcioglu, "A Study on the Number of Memory Ports in Multiple Instruction Issue Machines," *Proceedings of the 26th Annual International Symposium on Microarchitecture* (*MICRO-26*), 1993, pp. 49–58.
18. M. J. Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press, Cambridge, MA, 1989.
19. D. Bernstein and D. Cohen, "Dynamic Memory Disambiguation for Array References," *Proceedings of the 27th Annual International Symposium on Microarchitecture* (*MICRO-27*), 1994, pp. 105–111.
20. M. Moudgill and J. Moreno, "Run-Time Detection and Recovery from Incorrectly Reordered Memory Operations," *Research Report RC-20857*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1996.
21. K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill, "Dependence Flow Graphs: An Algebraic Approach to Program Dependencies," *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, 1991, pp. 67–78.
22. K. Ebcioglu, "Some Design Ideas for a VLIW Architecture for Sequential Natured Software," *Parallel Processing* (*Proceedings of IFIP WG 10.3, Working Conference on Parallel Processing*), M. Cosnard, M. Barton, and M. Vanneschi, Eds., 1988, pp. 3–21.
23. J. H. Moreno and M. Moudgill, "Scalable Instruction-Level Parallelism Through Tree Instructions," *Research Report RC-20661*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1996.
24. J. H. Moreno, "Dynamic Translation of Tree-Instructions into VLIWs," *Research Report RC-20505*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1996.
25. J. H. Moreno, M. Moudgill, K. Ebcioglu, E. Altman, B. Hall, R. Miranda, S. K. Chen, and A. Polyak, "Architecture, Compiler and Simulation of a Tree-Based VLIW Processor," *Research Report RC-20495*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1996.
26. T. Conte and S. Sathaye, "Dynamic Rescheduling: A Technique for Object-Code Compatibility in VLIW Architectures," *Proceedings of the 28th Annual International Symposium on Microarchitecture* (*MICRO-28*), 1995, pp. 208–218.
27. S. Vassiliadis, J. E. Phillips, and B. Blaner, "Interlock Collapsing ALUs," *IEEE Trans. Computers* **42,** 825–839 (July 1993).
28. S. Vassiliadis, B. Blaner, and R. J. Eickemeyer, "SCISM: A Scalable Compound Instruction Set Machine," *IBM J. Res. Develop.* **38,** No. 1, 59–78 (January 1994).
29. G. Kane, *PA-RISC 2.0 Architecture*, Prentice-Hall, Inc., Upper Saddle River, NJ, 1992.

Received August 8, 1996; accepted for publication March 18, 1997

Jaime H. Moreno *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (jmoreno@watson.ibm.com).* Dr. Moreno is a Research Staff Member in the Microsystems Department at the IBM Thomas J. Watson Research Center. He received a degree in electrical engineering from the University of Concepcion, Chile, in 1979, and M.S. and Ph.D. degrees in computer science from the University of California at Los Angeles in 1985 and 1989, respectively. In 1992 Dr. Moreno joined the IBM Research Division, where he has been performing research in processor architectures. Before joining IBM, he was a faculty member at the Department of Electrical Engineering, University of Concepcion, Chile, and collaborated as a postdoctoral researcher at UCLA. Dr. Moreno is co-author of the book *Matrix Computations on Systolic-Type Arrays*, Kluwer Academic Publishers, 1992. His research interests include processor architectures, instruction-level parallelism, application-specific systems, and simulation environments for the exploration of processor architectures. He is a member of IEEE and the IEEE Computer Society.

Mayan Moudgill *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (mayan@watson.ibm.com).* Dr. Moudgill is a Research Staff Member in the Microsystems Department. He received a B. Tech degree in computer science and engineering from the Indian Institute of Technology, Kanpur, in 1988, and M.S. and Ph.D. degrees in computer science from Cornell University in 1992 and 1994, respectively. Dr. Moudgill joined the IBM Research Division in 1994. His research interests include processor architectures, processor timers and simulators, and optimizing compilers.

Kemal Ebcioglu *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (kemal@watson.ibm.com).* Dr. Ebcioglu received the Ph.D. degree in computer science from the State University of New York at Buffalo in 1986, the M.S. degree in computer engineering from the Middle East Technical University, Ankara, in 1979, a Master of Music degree in composition from Ankara State Conservatory in 1977, and the B.A. degree in French and Romance languages and literature, Faculty of Letters, Istanbul University, in 1974. He joined the IBM Thomas J. Watson Research Center as a Research Staff Member in 1986, and is currently manager of the High-Performance VLSI Architectures group there. Dr. Ebcioglu has published extensively in the fields of architectures and compilation techniques for fine-grain parallelism (VLIW in particular), and algorithmic composition of tonal music. He is a Vice President of the International Federation for Information Processing (IFIP) Working Group 10.3 (Concurrent Systems).

Erik Altman *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (erik@watson.ibm.com).* Dr. Altman is a Research Staff Member in the Microsystems Department at the IBM Thomas J. Watson Research Center. He received bachelor's degrees in economics and computer science from MIT in 1983 and 1989, respectively, and M.S. and Ph.D. degrees in electrical engineering from McGill University, Canada, in 1991 and 1995, respectively. Dr. Altman's current interests include real-time dynamic compilation, VLIW, simulation, and instruction scheduling. He joined the IBM Research Division in 1995; his previous employment includes design of hardware for machine vision, real-time software for the automotive industry, and hardware and software for a decoy system for the Stinger-Post missile.

C. Brian Hall *IBM Canada Laboratory, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7, Canada, (cbhall@vnet.ibm.com).* Mr. Hall received his B.Sc. degree from Queen's University in Kingston, Ontario, in 1983 and his M.Sc. degree from the University of Toronto in 1986. He joined IBM in 1987 and has since worked in compiler development and code optimization. Mr. Hall is currently an Advisory Software Developer.

Rene Miranda *Cadence Design Systems, Inc., 270 Billerica Road, Chelmsford, Massachusetts 01824 (renem@cadence.com).* Mr. Miranda received his B.S. degree in computer science from the University of Maryland in 1986. He joined the IBM Research Division in 1993, working in the Mathematical Sciences and Microsystems Departments. Before joining IBM, he was a software developer in the EDA industry. He is currently a Senior Technical Staff Member of the Logic Verification group at Cadence Design Systems.

Shyh-Kwei Chen *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 712, Yorktown Heights, New York 10598 (skchen@watson.ibm.com).* Dr. Chen received a B.S. degree from the National Taiwan University in 1983, an M.S. degree from the University of Minnesota in 1987, and a Ph.D. degree from the University of Illinois at Urbana-Champaign in 1994, all in computer science. He joined the IBM Research Division in 1994, and worked on a parallelizing compiler for VLIW and superscalar architectures for two years. Dr. Chen is currently with the Servers Department, working on data-intensive platforms.

Arkady Polyak *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (arp@watson.ibm.com).* Mr. Polyak is a Staff Software Engineer in the Microsystems Department at the IBM Thomas J. Watson Research Center. He received a B.S. degree and an M.S. degree in computer science in 1987 and 1988, respectively, both from the Moscow Transportation University, Moscow, Russia, and an M.S. degree in operations research from New York University in 1997. He joined the Watson Research Center in 1990. Mr. Polyak's work has involved binary-to-binary translation, linkage, and simulation of various system architectures. His research interests include parallel architecture, performance analysis, and performance programming for scientific computation. He has co-authored several papers in the area of high-performance processors.