RC 20694 (91338) 11/14/96
Computer Sciences/Mathematics 10 pages

Research Report

Compiler/Architecture Interaction in a Tree-based VLIW processor

M. Moudgill, J.H. Moreno, K. Ebcioglu, E. Altman,
S.K. Chen, A. Polyak

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

Research Division
Almaden ● T.J. Watson ● Tokyo ● Zurich ● Austin

Compiler/architecture interaction in a tree-based VLIW processor

M. Moudgill, J.H. Moreno, K. Ebcioglu, E. Altman, S.K. Chen, A. Polyak

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

Abstract

This paper describes a compilation and simulation environment designed to explore the interaction among compiler and architecture for the case of a tree-based very-long instruction word (VLIW) processor. The environment is characterized by its flexibility and fast turn-around time, allowing the exploration of architecture/compiler trade-offs in several dimensions over complete execution runs of standard benchmarks. CHAMELEON, our research compiler, uses state-of-the-art optimizing techniques to extract and exploit instruction-level parallelism. FORESTA, the VLIW architecture, has an instruction set which is based on the PowerPC architecture. Results reported in the paper demonstrate the suitability of the environment for the purposes of evaluating trade-offs; in particular, the interactions arising from the availability of three-input instructions in the architecture are discussed. The exploration of such interactions has led to the development of some novel ideas in the architecture as well as in the compiler.

1. Introduction

The design of a new processor architecture and its associated compiler is a complex process. For a given set of requirements, designers must

- determine what attributes are important and necessary;
- design an architecture implementing such attributes; and
- fulfill implementation and cost constraints.

In general, the design process is an iterative one: features are proposed, required changes are introduced to the design environment, and the effectiveness/performance of the features is evaluated. Such a process, which requires adequate tools for simulation and performance measurement, is necessary due to the ever tighter interaction among compiler and architecture. 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.

We have been researching the viability of a very-long instruction word (VLIW) architecture in the context of PowerPC and the AIX operating system. We have been studying the potential features of such a new architecture for exploiting instruction-level parallelism (ILP), the appropriate compiler algorithms, and the interaction among compiler and architecture. The objective has been the development of a compiler/architecture which reaches new levels of ILP in branch-intensive programs. For these purposes, we have developed an experimental environment which provides reasonably fast turn-around time from compilation to simulation, so that compiler/architecture trade-offs are analyzed over complete execution runs [1]. Our tools have similar properties to those available in other simulation environments [2-3], but a combination of features make ours unique, including:

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

In this paper, we describe our environment for compiler/architecture interaction in the context of our target VLIW processor. We focus on features of 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 incorporate and evaluate new features. Quantitative

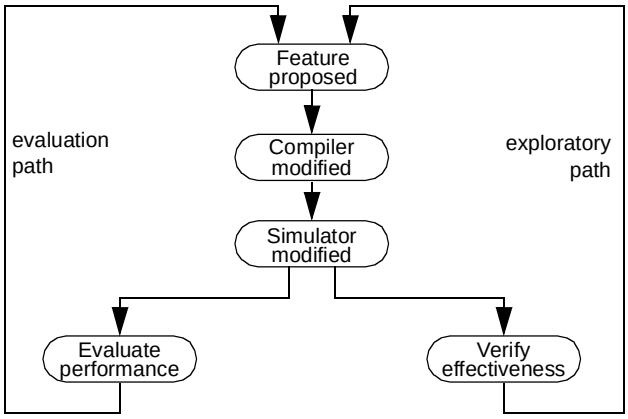

Figure 1: Iterative simulation/evaluation process

results reported illustrate the abilities (or limitations) of the compiler to exploit the architectural features considered. In practice, the environment allows evaluating alternative features over realistic workloads; programs such as the SPECint benchmark suite and a set of AIX utilities are simulated in their entirety. Simulation executables typically run only 10 to 15 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 turn-around time.

The rest of the paper is organized as follows. We first summarize relevant features of our environment. In Section 3, we briefly summarize the basic properties of FORESTA, our tree-based VLIW architecture, and in Section 4 we describe significant aspects of CHAMELEON, the optimizing compiler. Then, we describe architecture/compiler interactions which are possible, illustrating some of them with quantitative results. We finalize with some observations regarding the interactions among compiler and architecture as perceived in our context.

2. The development and simulation environment

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

- the *exploratory (fast) path*, which is characterized by fast turn-around time but only instruction-set architecture performance measurements; and
- the *evaluation (slow) path*, which is characterized by longer turn-around 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 processor implementation issues such as finite size caches, interlocks, and so on). In contrast, the evaluation path focuses on providing accurate performance estimates, including the implementation aspects.

The exploratory path has been built into a simulation environment which comprises two phases, as follows (see Figure 2):

Preparation phase, in which VLIW assembly language code 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).

Simulation phase, in which the VLIW program is simulated, including the collection of run-time profiling information.

This two-phase approach, which is common to many simulation/profiling tools [2], offers several special advantages in our case. For example, due to the same layout of data and procedure call conventions, the preparation phase allows mixing assembly code from the VLIW and PowerPC architectures. Since the translator generates PowerPC assembly

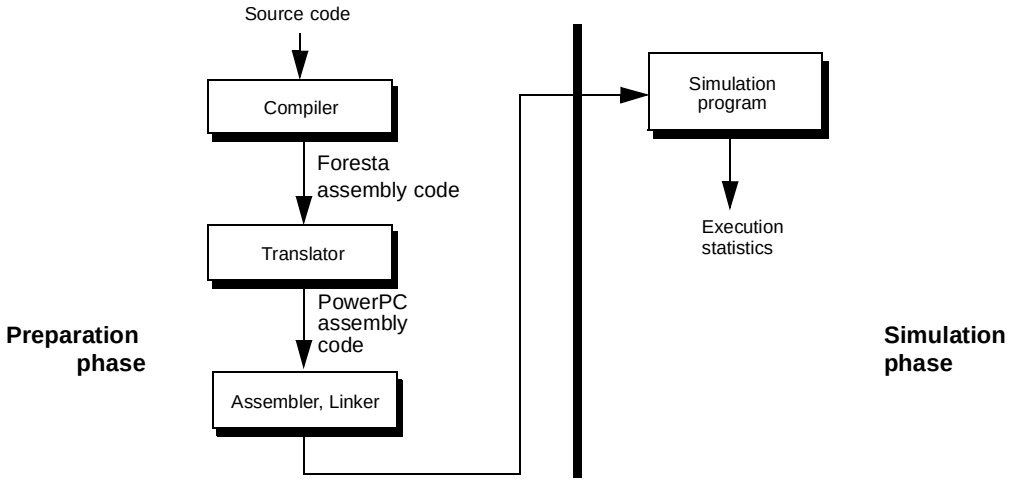

Figure 2: Overview of the simulation environment

code on a file-by-file basis, it is possible to compile into code for the VLIW architecture only a subset of the source files composing a program, and compile directly into PowerPC 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 FORESTA architecture, thus permitting focusing only on critical parts of a program.

As already stated, in this paper we do not address the features of the processor at the implementation level (e.g., alternative processor and memory organizations). However, implementation-related features are easily incorporated into our environment through specific program interfaces between a processor model, a memory model, and the emulated program. A complete description of the simulation environment, including the cycle-by-cycle timing capabilities, is given in [1].

3. FORESTA, a VLIW architecture based on tree-instructions

Branch-intensive programs can be conveniently represented as sequences of *tree-instructions* [4-5], 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 3).

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. All operations and the multiway branch are independent and executable in parallel.

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 destinations (registers or storage locations). In contrast, operations not on the taken path of the multiway-branch are inhibited from committing their results.

In FORESTA, our VLIW architecture [6], each tree-instruction is represented in main storage as a contiguous sequence of primitive operations which is obtained from the depth-first traversal of the tree; in contrast to traditional VLIW processors, this representation does not use no-ops in the encoding of programs. All possible next tree-instructions for a given tree are stored as a block in adjacent memory locations. This representation allows achieving binary compatibility among different implementations of this architecture, each with varying degrees of parallel execution capabilities, by allowing the

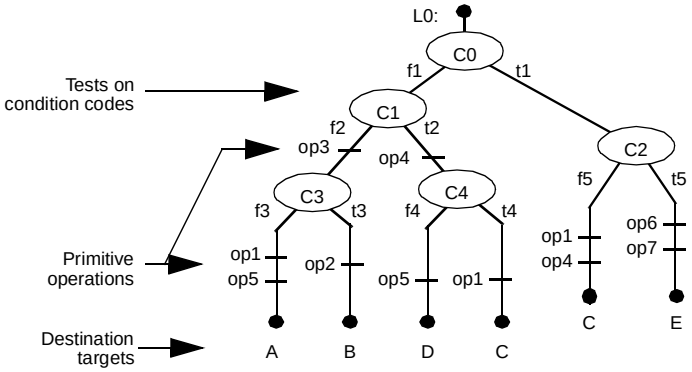

Figure 3: A tree-instruction

dynamic decomposition of a large tree into subtrees which are executed in different cycles [7].

The primitive operations in FORESTA are based on the PowerPC architecture [8]; deviations from the PowerPC instruction set include:

- larger register set than the 32 general-purpose, 32 floating-point, 8 condition registers* available in the PowerPC architecture;
- support for speculative non-trapping load operations;
- some complex operations have been deleted, including rotate-and-mask, update form and indexed load/store, load/store multiple, and string operations;
- the "record" form of operations allows specifying any of the condition registers (instead of the implicit Condition Register 0);
- the displacement field in memory operations has been reduced from 16 to 11 bits;
- the encoding of some operations has been extended to 64 bits;
- some operations can support 32-bit immediate fields;
- some 3-input fixed-point operations have been added, such as add&shift, and∨
- some support for conditional execution has been added, in the form of conditional move and conditional store operations.

* We treat the condition register fields in the PowerPC Condition Register as separate registers.

4. 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, which was designed to target VLIW architectures that execute tree-instructions, is extensively parameterized so that it can target processors with different features, such as issue width, number of functional units, instruction latencies, register set, and so on. Supporting architectural explorations and implementing aggressive optimizations geared towards several different targets implies a compiler in constant state of change. Consequently, the compiler has been designed with support for adding code, verifying the modified compiler, and isolating problems quickly (such mutability has led to 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 [9], 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 FORESTA 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 FORESTA processor, or is translated into PowerPC assembly code.†

The modifications to ***xlc*** and ***gcc*** fall into two categories: first, we have turned off phases such as scheduling and loop transformations that would

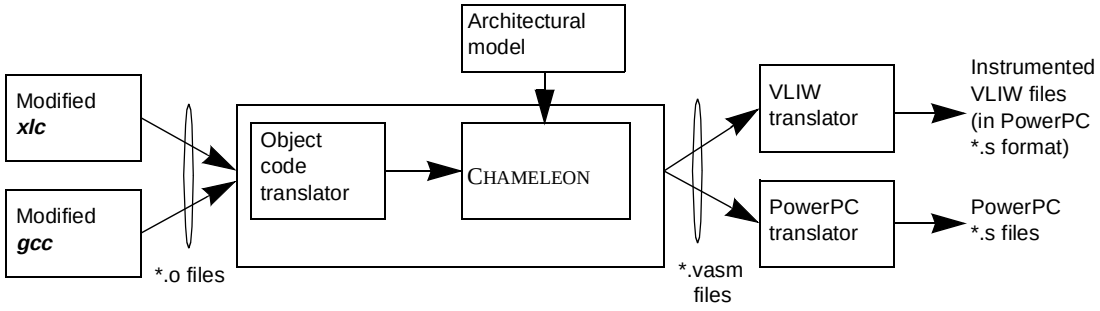

Figure 4: The CHAMELEON compiler environment

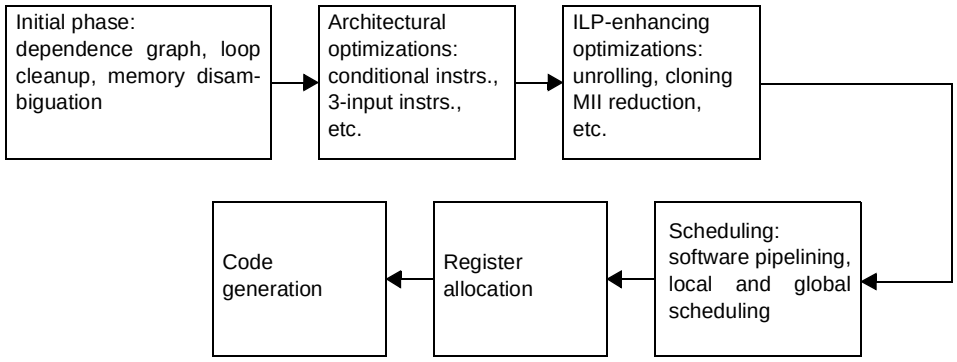

Figure 5: Phases in CHAMELEON

tend to obscure the original code sequence; second, we have added capabilities to convey information such as alias classes, registers live at function call and function return points, spill locations, etc. 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 front-end.

Optimizations

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

Traditional. These include constant propagation, loop invariant code motion, dead-code elimination, and sub-expression elimination. These optimizations are applied throughout the compilation process. Moreover, since we use the output from ***xlc***, we take advantage of ***xlc***'s excellent set of traditional optimizations [10].

ILP-increasing. Optimizations designed for increasing the instruction-level parallelism, so that the scheduler can pack instructions more tightly. These include various loop transformations, such as unrolling, re-writing loops with commutative/associative operations (reductions), re-writing 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 inter-procedural 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 [11]. We take no advantage of inter-procedural analysis;

† The compiler can also be restricted to directly produce primitive operations complying with the PowerPC architecture; such code is suitable for a wide-issue implementation of the PowerPC architecture.

even the inter-procedural phase of ***xlc*** has been disabled.

The scheduler used by CHAMELEON is an enhanced version of selective scheduling [5]; as a result, all loops are subject to software pipelining, including those containing multiple paths and other loops. The original selective scheduling 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).

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) [12], provides an integrated data/control flow information well suited for incorporating advanced optimizations; such optimizations are, with few exceptions, independent of each other and permutable.

CHAMELEON is extensively parameterized; everything from the processor resource model to the instruction set is defined through tables, which are usually modifiable at runtime. 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.

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.

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 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!).

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's 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.

5. Some examples of compiler/architecture interactions

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

- 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;

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

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

(a) Register set

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

(b) Instruction latencies

Figure 6: Size of register set and latencies in primitive operations

- 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 describe two of these features in detail, namely the exploration of issue-width with larger register set, and the incorporation of three-input operations in the architecture.

5.1 Instruction-level parallelism

Typically, studies on VLIW architectures assume a fixed size register set and investigate the effects of increasing the operations per VLIW [5][13-14]. In addition to such studies, we have explored the availability of instruction-level parallelism assuming larger register set for wider-issue implementations. Such trade-offs are 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).

We present results for three widths of a VLIW processor that uses all the instruction-set architecture extensions described earlier; the three widths are, respectively, capable of issuing any 8, 12 and 16 instructions per VLIW. The size of the register sets considered are listed in Figure 6a; the experiments reported have been performed using the operation latencies listed in Figure 6b. The programs used are from the SPECint 92 and 95 suites.

Table 1 reports the instruction-level parallelism obtained in the VLIW code for different processor configurations. This table indicates the ratio between the number of instructions executed by a RS/6000 processor running PowerPC code (i.e., with ILP=1) to the number of tree-instructions executed by the FORESTA processor, for the different issue-widths. The PowerPC instruction counts used for these ratios are obtained from compiling the programs with ***xlc*** at optimization level -O2.

Table 1: VLIW instruction ratio with respect to sequential code

Benchmark	Issue-width		
	8	12	16
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
m88ksim	2.70	3.02	3.04
go	2.23	2.45	2.51

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 [10]. 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. As can be inferred from Table 1, even under this stringent condition, the improvement in ILP is comparable to that previously reported in the literature.

5.2 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 [15-16]. 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 [17] have some capabilities of this type. Note that the potential benefit of adding three-input instructions 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 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 the pair is not in the critical path of the program.

We have explored the benefits of including combined operations in our 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;* and
L: any combination of logical with logical operations.

Moreover, for determining an upper-bound on the potential performance achievable, we have 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. Due to encoding constraints, these combinations require using a double-word 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 (six lines), 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 so that it properly handles the single-cycle latency of a three-input operation, thereby combining instructions when such an action would produce a better schedule; and
- altered the final peephole compaction phase so that combining adjacent VLIWs also recognizes and exploits the three-input operations.

Table 2 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 8 memory operations, 4-way branch, and 2 floating-point operations, and whose register set is 64/64/16 registers. As listed in the table, only some programs exhibit significant gains, whereas other reflect little variation.

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

Benchmark	Base ILP	3-in 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 %

Table 3 through Table 5 illustrate the distribution of the most common three-input operations for benchmarks *compress*, *espresso* and *go*, respectively; these benchmarks are the only ones which exhibit real gain from the availability of the new instructions. In these tables, the static instruction count represents the number of occurrences of the specific instruction combinations in the VLIW program, whereas the dynamic ratio corresponds to the ratio among the dynamic count of the specific instruction combina-

* This class does not include arithmetic shift.

tions to the total operations in the entire program (specified either in the taken or in non-taken paths of the tree-instructions).

Table 3: Distribution of three-input operations in benchmark compress

Operation	Static instr. count	Dynamic ratio
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 4: Distribution of three-input operations in benchmark espresso

Operation	Static instr. count	Dynamic ratio
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 %

Table 5: Distribution of three-input operations in benchmark go

Operation	Static instr. count	Dynamic ratio
slw,add	10393	11.6 %
add,add	947	0.4 %
add,subf	341	0.2 %
slw,subf	83	< 0.01 %
srw,add	1	< 0.01 %

As a final example of interaction among compiler and architecture, Table 6 depicts the effects of the three-input instructions grouped according to their classes, for the same three benchmarks as above. 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.

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

Instrs. 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 %

6. Concluding remarks

We have described our environment for studying compiler/architecture interactions, in the context of the FORESTA VLIW architecture, and have illustrated it through the analysis of the effects arising from the availability of three-input instructions in the architecture.

The CHAMELEON compiler uses state-of-the-art techniques to reach new levels of ILP in branch-intensive programs. The compiler and the simulation environment have been developed with the objective of supporting the interaction among compiler and architecture, and have been designed primarily for mutability. The run-time performance of the compiler has been sacrificed in those places where it would have inhibited its suitability for modifications.

The simulation environment provides fast turn-around time from compiler output to simulation results, allowing rapid testing of new compiler algorithms and new architecture features. Simulation executables typically run only 10 to 15 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 store execution traces or other simplifications to reduce turn-around time.

In practice, the environment is allowing us to evaluate alternative architecture/compiler features over realistic workloads. Programs such as the SPECint 92 and 95 benchmark suites and a set of AIX utilities are being simulated in their entirety, for different processor configurations and different compiler algorithms.

Acknowledgments

We thank Brian Hall, Norman Cohen, Richard Goldberg, Peter Oden, and Balaram Sinharoy for their contributions to different parts of CHAMELEON.

References

[1] J.H. Moreno et al., "Architecture, compiler and simulation of a tree-based VLIW processor," Technical Report RC-20495, IBM T.J. Watson Research Center, July 1996.

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

[3] 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.

[4] 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.

[5] 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.

[6] J.H. Moreno et al., "FORESTA User Instruction Set Architecture," Technical Report (in preparation).

[7] J.H. Moreno, "Dynamic translation of tree-instructions into VLIWs," Technical Report RC-20505, IBM T.J. Watson Research Center, July 1996

[8] IBM Corporation, *PowerPC Architecture*, 1st. edition, 1993.

[9] IBM Corporation, *AIX XL C compiler*, IBM 1993.

[10] M. Auslander, M. Hopkins, "An overview of the PL.8 compiler," in *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pp.22-31, June 1982.

[11] T. Ball, J. Laurus, "Branch prediction for free," in *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation*, pp. 300-313, June 1993.

[12] 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

[13] 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.

[14] 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.

[15] S. Vassiliadis, J.E. Phillips, B. Blaner, "Interlock collapsing ALUs," *IEEE Transactions on Computers*, vol. 42, pp. 825-839, July 1993.

[16] S. Vassiliadis, B. Blaner, R.J. Eickemeyer, "SCISM: a scalable compound instruction set machine," *IBM Journal of Research and Development*, vol. 38, no. 1, January 1994.

[17] Hewlett-Packard Company, *PA-RISC 1.1 architecture and instruction set reference manual*, 2nd. edition, 1992.