RC 20661 (91417) 9/12/96
Computer Sciences/Mathematics

IBM Research Report

Scalable instruction-level parallelism through tree-instructions

Jaime H. Moreno, Mayan Moudgill
jmoreno@watson.ibm.com, mayan@watson.ibm.com

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

LIMITED DISTRIBUTION NOTICE
This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties).

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

Scalable instruction-level parallelism through tree-instructions

Jaime H. Moreno, Mayan Moudgill
(jmoreno@watson.ibm.com, mayan@watson.ibm.com)

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

Abstract

We describe a representation of instruction-level parallelism which does not require checking dependencies at run-time, and which is suitable for processor implementations with varying issue-width. In this approach, a program is represented as a sequence of *tree-instructions*, each containing multiple primitive operations and executable either in one or multiple cycles, but which do not require a specific processor organization. Tree-instructions are generated assuming an implementation capable of large instruction-level parallelism. During instruction cache reloading/accessing, tree-instructions are decomposed into subtrees which fit the actual resources available in an implementation, and which require a simple instruction-dispatch mechanism as in the case of statically scheduled processors. The representation makes practical the use of the same parallelized code in implementations with different issue-width; simulation results indicate that the instruction-level parallelism achievable with this approach degrades less than 10% with respect to code compiled for each specific implementation.

1. Introduction

Modern computer architectures exhibit object-code compatibility, that is, the capability to execute the same object code on all implementations of the architecture. Although members of a given processor family may differ in their resources and capabilities for dispatching primitive operations, all processors can execute the same object code. The processors rely on features such as dynamic scheduling of primitive operations, interlocking, register renaming, and branch prediction to exploit instruction-level parallelism (ILP) in programs and take advantage of the resources available [1]. However, for best performance, object code is usually tuned to the specific resources in an implementation.

Very-long instruction word processors have been proposed as an alternative to dynamically scheduled processors for exploiting instruction-level parallelism. These processors contain multiple functional units, fetch from the instruction cache a very-long instruction word (VLIW) which specifies the operations performed in each functional unit, and dispatch the entire VLIW for parallel execution. These capabilities are exploited by compilers which generate code that has grouped together independent primitive instructions executable in parallel (static scheduling). Processors have relatively simple control logic because they do not perform dynamic scheduling nor reordering of operations, as done by dynamically scheduled processors [1].

VLIW object-code such as that used in [2-4], is explicitly dependent on the features of the implementation (i.e., the number, type and location of the functional units). This has been a conscious decision, under the assumptions that the implementations would be simpler, and the compiler/programmer could better exploit the processor resources if he/she had good knowledge of its features and limitations. In other words, the separation among *architecture* and *implementation*, that has been common practice in processor design for scalar/superscalar implementations has been sacrificed in VLIW implementations, in order to simplify and exploit better the hardware by the compiler/programmer. On the other hand, VLIW-based compilation techniques are geared towards extracting ILP and representing it explicitly; such ILP has also been shown suitable for dynamically scheduled processors [5].

The downside of explicit ILP is that it either sacrifices compatibility among implementations, or is tuned for a single implementation. Object code with explicit ILP usually contains the detailed organization of one specific implementation, which might be different from others, and/or uses features that might not exist in all implementations. The cost associated with porting and maintaining code across implementations has limited the acceptability of the explicit ILP paradigm as part of a processor architecture [6-8].

Two classes of solutions have been pursued to solve the compatibility/tuning problem in the case of explicit-representation of ILP: software-based and hardware-based techniques. Software-based approaches (excluding source compilation due to its inherent limitations) rely on binary-to-binary (object-code) translation, either with or without hardware support [9-12]; some of these actually correspond to translating binary code among different architectures, but they can be applied to different implementations of the same architecture.

Of particular relevance is the scheme described in [12], which addresses the translation of code from one VLIW implementation to another by performing code rescheduling at page-fault time. This is a promising general technique, capable of coping with any implementation constraint; its limitations are the overhead introduced at page-fault time and the added complexity required to manage rescheduled and non-rescheduled pages. In contrast to other solutions, the scheme assumes that the object-code is already an explicit representation of ILP, and is able to exploit that representation accordingly.

A hardware-based solution for VLIW processors was proposed in [13], wherein the concept of *delayed split-issue* together with dynamic scheduling hardware are introduced; these characteristics allow using the interlocking and scoreboarding techniques known for dynamic scheduling of instructions. In fact, the primary objective of that paper is "to show that dynamic scheduling is as viable for VLIW processors as with more conventional ones." As a result, the object-code generated for one implementation of a VLIW processor family can be executed on an implementation with fewer functional units and/or different latencies for the operations, at the cost of hardware complexity.

Another related class of hardware-based solutions consists of dynamic extraction of ILP and its reuse. For example, the *fill-unit* approach originally proposed in [14] and extended in [15], as well as the scheme described in [16], extract VLIWs from the sequential execution of a program and saves them, so that successive executions of the same instructions are performed using the parallel rather than the original sequential version. Although promising, this type of approaches is limited due to the small "window" of instructions analyzed, and the complexity associated to run-time parallelizing.

In this paper, we describe an explicit representation of instruction-level parallelism which does not require complex dependence checking at run-time nor a specific processor organization, but which is suitable for processor implementations with varying issue-width. Programs are represented as sequences of *tree-instructions* generated assuming an implementation capable of large instruction-level parallelism. Implementation-specific aspects, including restricted ILP capabilities, are introduced during instruction cache reloading/accessing, and the simplicity in instruction-dispatch logic that is characteristic of statically-scheduled processors is preserved. Basic objectives of this approach are:

- object-code generated for any implementation of an architecture can be executed in all implementations;
- performance should increase when the same object-code is executed in a larger implementation (within a reasonable range); and
- the ILP achieved on a particular implementation should not degrade severely with respect to compilation of the source program for that specific implementation.

We first describe our proposed explicit representation of instruction-level parallelism, then a mechanism for translating the program representation into one suitable for execution in a processor with fewer ILP capabilities than those assumed at program generation. We focus on instruction-set architecture aspects, that is, on the representation of ILP rather than on the implementation issues. We present simulation results using the proposed representation, which indicate that code compiled for a large implementa-

tion but executed on a smaller one degrades less than 10% with respect to code compiled for the smaller implementation. Such a small degradation makes practical the use of the same object code in a family of processors, with ILP gains proportional to the issue-width of those processors.

2. Program representation based on tree-instructions

The objective of the approach described in this paper is providing statically-scheduled object-code which delivers scalable performance across a range of implementations. For this purpose, we assume that the code generated by the compiler/programmer is targeted for maximum parallelism; that is, the object-code contains as much instruction-level parallelism as is possible to extract from a program, subject to limitations on code explosion, code speculation, and related issues. Although such code would deliver best ILP on an implementation matching the maximum parallelism expressed in it, if that implementation was available, only minor degradation (with respect to compiled code) is introduced when the code is executed in a smaller implementation.

To achieve the objective stated, we propose representing a program as a sequence of *tree-instructions*, or simply *trees*, each of which corresponds to an *unlimited multiway branch* with multiple branch targets and an *unlimited set of primitive operations* (see Figure 1). All operations and branches are independent and executable in parallel. The multiway branch is associated with the internal nodes of the tree, whereas the operations are associated with the arcs. The multiway branch is the result of a set of *binary tests on condition registers*: the left outgoing arc from a tree node corresponds to the false outcome of the associated test, and the right outgoing arc corresponds to its true outcome. These tree-instructions are an extension of those described in [17-19].

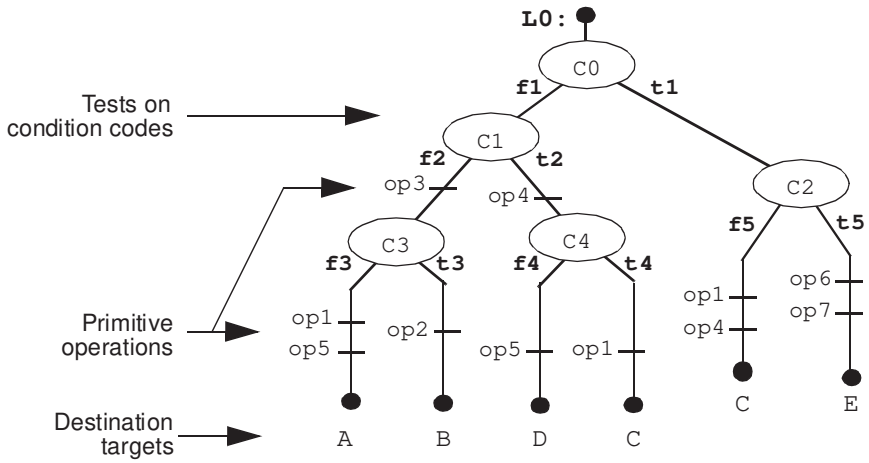

Figure 1: A tree-instruction

Based on the evaluation of the multiway branch, a single path within a tree-instruction is selected at execution time as the *taken path* (a tree-path starts from the root of the tree and ends in a branch target). Operations on the taken path are executed to completion, and their results placed in the corresponding target registers or storage locations. In contrast, operations not on the taken path of the multiway-branch are inhibited from committing their results to storage or registers; such operations do not produce any effect on the architectural state of the processor. Execution paths are assigned to tree paths in left-to-right order, according to their probability of being taken: the most probable path is the left-most one whereas the least probable path is the right-most one.

Primitive operations are subject to *sequential constraints for each tree-path.* That is, the final result from executing a tree must be the same as if all primitive operations in the taken path are executed one at a time

(sequentially). Consequently, a primitive operation cannot use a resource which is set by a previous operation in the same tree-path (a read after write conflict); Figure 2 illustrates legal and illegal tree-instructions.

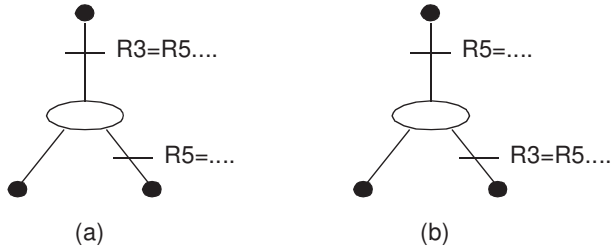

Figure 2: Legal (a) and illegal (b) tree-instructions

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; for example, the code sequence in Figure 3 represents the 6-way tree-instruction depicted in Figure 1. Testing a condition register is performed with a *skip conditional* operation, which corresponds to a flow-control operation within the tree, and which indicates where the tree-path corresponding to the true outcome of the test continues in storage; as a result, a skip conditional operation is a branch with a (short) positive displacement. On the other hand, the branch targets of a tree are represented as *unconditional branch* operations, which specify the next tree to be executed when the corresponding path of the current tree is selected. The end of a tree is implicitly delimited by a primitive operation that follows an unconditional branch which is not reachable by any skip conditional operation within the tree; for example, the end of the tree in Figure 3 is detected at label `L1` because such a label does not appear in any of the skip operations.

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

Figure 3: Sequential representation of a tree-instruction

3. Decomposition of tree-instructions

Tree-instructions can be executed in implementations with varying degrees of parallel execution capabilities, without requiring complex hardware for the detection of dependencies or scheduling of instructions. The entire tree can be executed simultaneously, if there are sufficient resources in the processor. On the other hand, if a tree-instruction exceeds the resources available in an implementation (such as branching degree, number of fixed-point or floating-point units), then the tree-instruction can be decomposed *(pruned)* into subtrees which are executed in different cycles; this is possible thanks to the sequential constraints in each tree-path. For example, the 6-way tree-instruction depicted in Figure 1 can be executed in a

processor capable of at most a 4-way branch by pruning the tree at the first two skip operations, as depicted in Figure 4; the original tree-instruction is decomposed into three smaller subtrees, the first one containing a 4-way branch (targets A,B,T2,T1), and the other two containing 2-way branches (targets D,C and C,E, respectively).

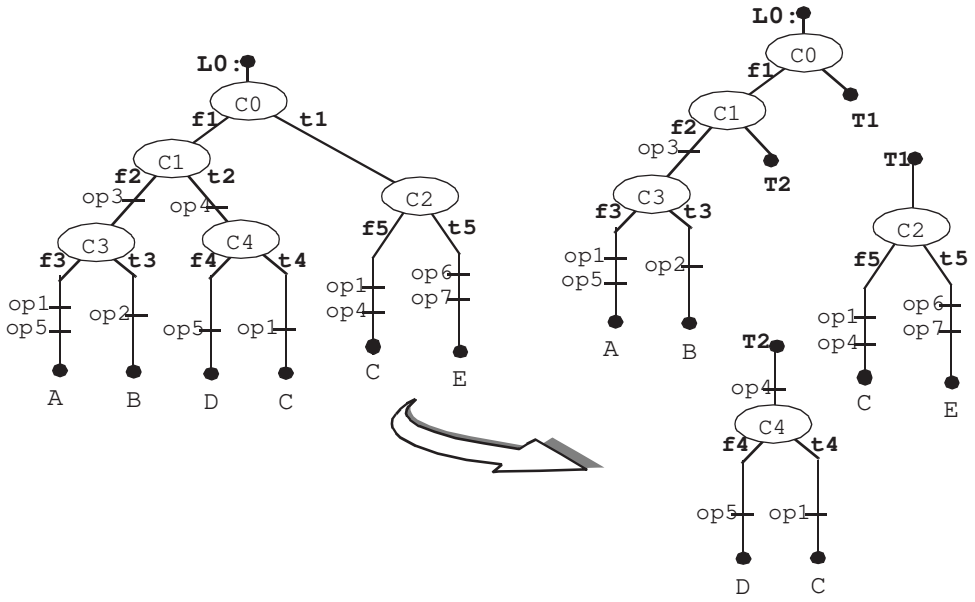

Figure 4: Pruning a tree-instruction

The subtrees resulting from the decomposition process have the same general structure as the original trees (that is, a multiway-branch tree with operations in the tree-paths), but their size is limited. Subtrees are executed in successive cycles, until the taken path within the original tree is completely traversed. No dependence checking or instruction scheduling is required, but only the determination of the end of the tree and the structure of the resulting subtrees; these tasks are much simpler to achieve.

Pruning degrades ILP with respect to the execution of complete trees whenever the taken path is not the most probable one, and whenever the number of operations in the most-probable path exceeds the resources available in an implementation. On the other hand, as long as the taken path is fully contained within the first subtree, no ILP degradation occurs because the remaining subtrees are not executed. Evidently, if a program is generated assuming a given degree of instruction-level parallelism but an implementation is capable of more than that, then there is neither performance gain nor performance degradation with respect to execution on an implementation with exactly the degree of parallelism assumed at program generation.

Tree-instructions are decomposed into subtrees in a two-step process, as follows:

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

   For example, Figure 5 illustrates the I-cache representation of the subtrees depicted in Figure 4, assuming 32 primitive operations per cache line, with the first subtree starting at offset 4 within the line. In this figure, each square corresponds to one operation, and the extra information added into the cache

line is represented by the different patterns and the vertical bars; only a few extra bits per primitive operation are required in the I-cache. The target addresses of the pruning points (the arrows in the figure) are not needed explicitly because those targets are matched positionally with the pruning points: the last pruning point is associated with the first subtree after the end of the subtree containing the pruning point, the next subtree with the previous pruning point, and so on.

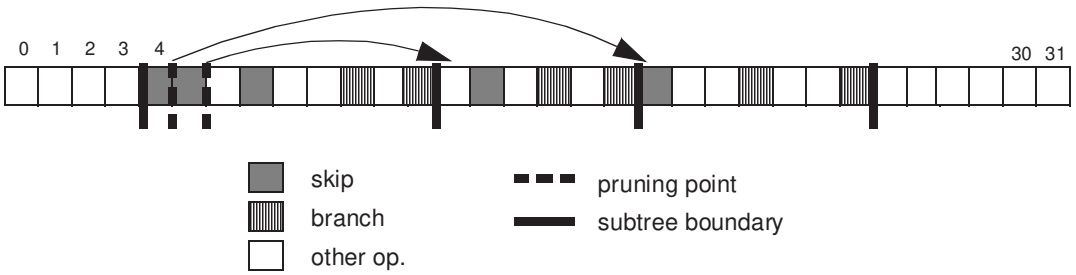

Figure 5: Instruction cache representation of subtrees

2. **Masking step.** At I-cache access time, the subtree starting at the specified memory address is fetched. Exactly as many primitive operations as the maximum that the processor is capable of executing at once are fetched; if the subtree contains fewer operations than those fetched from the I-cache, then those operations beyond the end of the subtree are disabled (masked out), as if they were a path that is never taken.

The pruning step, which increases the I-cache miss penalty by an amount that depends on the implementation, is described by the pseudo-code depicted in Figure 6. Basically, the procedure consists of fetching a block from main memory and inspecting each primitive operation in the block. For each operation, it is added into the current subtree if there are resources available of the corresponding type; otherwise, the tree-instruction is pruned at that point, completing the current subtree. If no pruning occurs, the tree is completed when the number of unconditional branch operations matches the number of paths as determined by the number of skip operations (each skip operation adds a path).

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

Figure 6: Pruning algorithm

As described above, the pruning step allows subtrees starting at any primitive operation boundary and of any length, as well as subtrees that straddle cache line boundaries. Since such a flexibility might lead to implementation complexities, some restrictions might be required to simplify a realization. For

example, an implementation of the pruning step which analyzes four primitive instructions per cycle is described in [20]; that implementation restrict the alignment of subtrees to quadword boundaries, and trees are pruned at cache line boundaries.

4. Execution of subtrees

The execution of a subtree consists of the following tasks:

1. evaluating the multiway branch specified in the subtree and selecting the path taken;
2. executing all operations contained in the subtree;
3. dispatching to the instruction cache the address of the next subtree to be executed; and
4. completing all operations in the taken path of the multiway branch, discarding the effects of operations in other paths.

The evaluation of the multiway branch generates two results:

- the address of the next subtree to be executed; and
- an *execution mask (EM)* identifying the operations within the subtree which appear in the taken path, that is, indicating which operations should complete their execution by placing their results in the corresponding destinations, and which operations should be aborted (their results discarded). For example, this mask could specify one bit per operation enabling/disabling its completion.

Figure 7a depicts execution masks for each path of the tree-instruction shown in Figure 3, for a processor capable of executing subtrees with up to 24 primitive operations and a 6-way branch, so that the entire tree can be executed at once (each execution mask is 24 bits long). In contrast, Figure 7b illustrates the execution masks for the case of a processor capable of executing subtrees with at most 16 operations and up to a 4-way branch (each mask is 16 bits long); the subtree corresponds to the first subtree depicted in Figure 4. In both cases, there are operations beyond the end of the subtree which are fetched from the I-cache but are disabled from execution (represented by the right-most bits -in boldface- set to 0).

Path	Execution mask
L0-A	111111100000000000000000
L0-B	111100011000000000000000
L0-D	110000000111100000000000
L0-C	110000000110011000000000
L0-C	100000000000000111100000
L0-E	100000000000000100011100

(a)

Path	Execution mask
L0-A	11111110000000
L0-B	11110001100000
L0-T2	11000000000000
L0-T1	10000000000000

(b)

Figure 7: Execution masks for paths in Figure 3 and Figure 4

Since the memory representation of the tree-instructions is obtained from the depth-first traversal of the tree, the process for generating the masks is quite simple. The execution mask associated to the first path has its bits set to 1 starting at the first operation up to the first branch operation; all other bits are set to 0. The mask associated to the second path has bits set to 1 starting at the first operation up to the last

skip operation before the first branch operation, and from the first operation after the first branch up to the second branch; all other bits are set to 0. The other execution masks are generated following the same principles. A detailed example implementation of this process is given in [20].

5. Experimental evaluation

We now present simulation results on the ILP degradation in the approach proposed here with respect to code statically scheduled specifically for the different implementations. The workloads used for this evaluation are listed in Figure 8; these are a set of workloads from the SPECint92 benchmark suite, and a collection of utilities in the AIX* operating system. Workloads such as these are usually chosen as representative of typical non-numeric user environments.

SPECint92	Description
compress	Compression/decompression utility
eqntott	Truth table generator for logic circuits
espresso	Boolean function minimization
xlisp	Lisp interpreter

AIX utils.	Description
fgrep	Regular expression matching on a file
lex	Lexical analyzer
prof	Runtime profile generator
sed	Stream editor
yacc	Parse generator

Figure 8: Workloads used in experimental evaluation

The primitive operations assumed available are based on the PowerPC architecture [21]; deviations from the PowerPC instruction set include:

- 64 general-purpose, 64 floating-point, 16 condition registers†;
- support for speculative non-trapping load operations;
- some complex operations have been deleted, including rotate-and-mask-insert, 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 present results for processors whose organizations are given in Figure 9, assuming two floating-point units in all cases. The simulation environment used and its capabilities are described more extensively in [22-23]. For the experiments in this paper, the environment basically consists of

- CHAMELEON, our optimizing research compiler which generates tree-instructions in an assembly code;
- a *pruner*, which transforms the original tree-instructions into subtrees that fit a particular implementation; and
- a *simulator*, which emulates the functionality of a particular processor implementation and contains instrumentation to collect execution data.

* AIX, RS/6000 are trademarks of International Business Machines Corporation

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

Configurations	Functional units			
	Integer	Memory	Branch	Maximum
A (16/8/8/2/16)	16	8	8	16
B (12/6/6/2/12)	12	6	6	12
C (12/4/4/2/12)	12	4	4	12
D (8/4/4/2/8)	8	4	4	8
E (8/4/2/2/8)	8	4	2	8
F (6/3/3/2/6)	6	3	3	6
G (6/3/2/2/6)	6	3	2	6

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

Figure 9: Processor configurations in experimental evaluation

Pruning has been implemented as a separate tool, prior to invoking the simulator. The pruner accepts tree-instructions generated by the compiler, and decomposes into subtrees those which exceed the resources available in the implementation; the result is a new program which fits the implementation. Although this is a static process, it has the same functionality as the dynamic one that would be performed by the pruning unit in an implementation; the object code executed in both cases is identical.

The methodology used for the evaluation consists of counting the number of instruction cycles required in the following cases (assuming that each subtree is dispatched in one instruction cycle, and that primitive operations are scheduled statically respecting their latencies):

a. code compiled according to the resources of the implementations; and

b. the same program used in the largest configuration in (a) on implementations with fewer resources.

In other words, we compare the instruction count of programs compiled to match the resources in an implementation with the instruction count of programs pruned to fit into smaller ones.

The capabilities of CHAMELEON regarding extraction of instruction-level parallelism are illustrated in Table 1 and Figure 10, which indicate the ratio among the number of instructions required by a PowerPC processor with ILP=1 to the number of tree-instructions required by the wide-issue implementations with the instruction set extensions listed earlier. CHAMELEON generates tree-instructions using synthetic branch probabilities produced by a variant of the Ball-Larus heuristics [24]; no profiling-directed feedback is used. The PowerPC instruction counts used for these ratios are obtained from compiling the programs with ***xlc*** [25], the IBM optimizing C compiler, at optimization level -O2. Note that we compute this ratio 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 in this case are obtained using the best compiler available for the PowerPC architecture. As can be inferred from Table 1, even under this stringent condition, the improvement in ILP is comparable to the best values previously reported in the literature.

As it could be expected, ILP decreases for smaller configurations; for example, the ILP of *compress* in the smallest configuration is half of the ILP in the largest configuration. Similar results are obtained for the other workloads; the most sensitive one is *eqntott*, whereas the least sensitive is *sed*. As it could also be expected, the ILP gain from increasing the issue-width of the processor is attractive but sublinear; doubling the number of resources produces an increase in ILP lower than 2. The results also show that the ILP achieved is sensitive to the type of units, with multiway branching an important one; for certain branch intensive programs, the 6/3/3/2 configuration yields better performance than the 8/4/2/2 configuration

Table 1: Instruction ratio with respect to sequential code

	Configuration						
Benchmark	**A**	**B**	**C**	**D**	**E**	**F**	**G**
compress	5.19	4.50	4.30	3.47	3.14	2.86	2.56
eqntott	7.67	4.91	4.91	3.50	3.35	2.67	2.65
espresso	2.86	2.68	2.68	2.41	2.03	1.98	1.81
fgrep	5.72	5.69	5.63	3.01	2.04	2.41	2.03
lex	4.31	3.15	3.13	3.39	1.95	2.37	1.90
prof	2.28	2.35	2.20	2.00	1.83	1.70	1.59
sed	3.07	3.07	3.05	3.04	2.51	2.66	2.37
xlisp	3.50	3.30	3.06	2.79	2.44	2.33	2.18
yacc	3.36	3.09	3.05	2.60	2.16	2.13	1.93

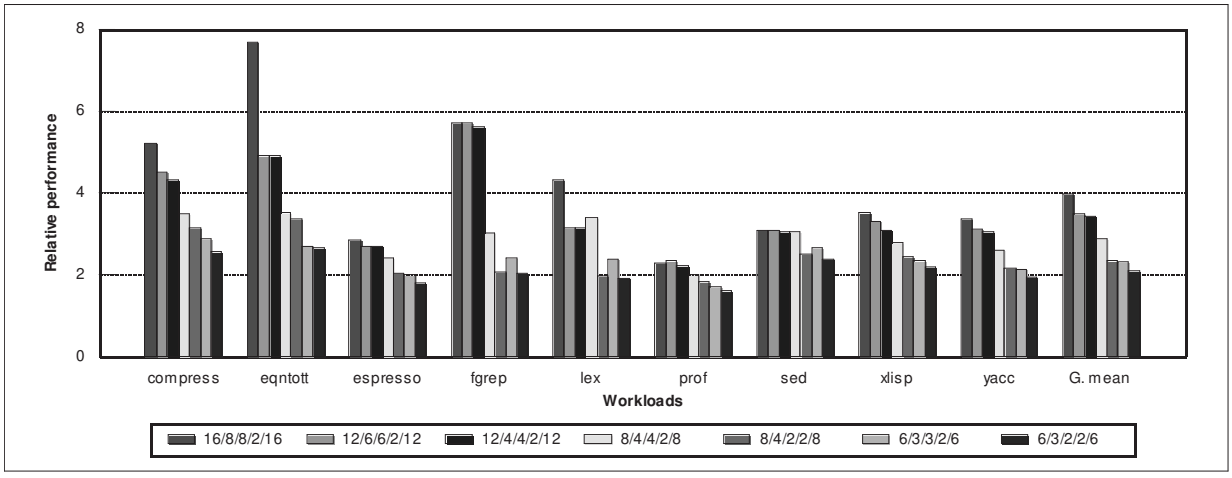

Figure 10: Relative performance of compiled code

because it supports more branches. In a couple of cases, narrower configurations yield better results than wider ones; this is due to the vagaries of the compiler algorithms, which we continue exploring.

Figure 11 illustrates the relative ILP of the configurations using in all cases the object-code generated for the largest configuration. Figure 12 compares the relative ILP of the pruned code with respect to the compiled code (ratio among the data in Figure 10 and Figure 11). These figures show that in many cases the ILP achieved with pruned code is about the same as the one obtained from code compiled for the spe-

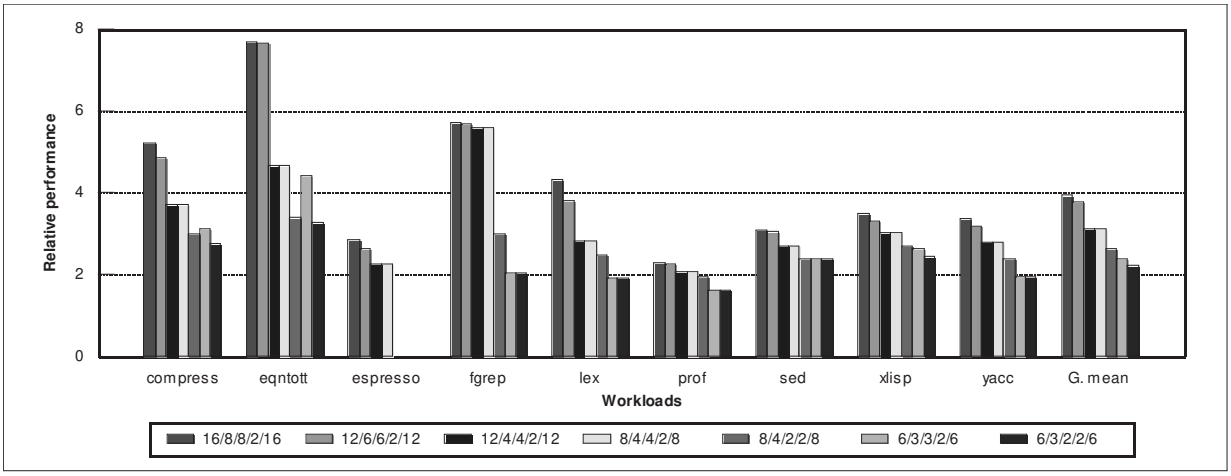

Figure 11: Relative performance of pruned code

cific implementation; in some other cases the degradation arising from pruning is not severe (less than 10%). Interestingly enough, there are some cases in which the pruned code actually delivers higher ILP than the code compiled for the specific implementation, sometimes significantly better. Among other reasons, this phenomenon is a consequence of the discrete nature of the current implementation of some optimizations in the compiler. For instance, a loop which requires eight functional units is unrolled twice for the 16-issue configuration, whereas the same loop is not unrolled for the 12-issue configurations thus leaving some units idle; a more "continuous" implementation of the compiler algorithms would unroll the loop three times and achieve three iterations every two tree-instructions. The amount of ILP exposed by aggressive speculation algorithms used for wide-issue implementations is also a factor. We continue investigating this behavior.

The experimental results indicate that the proposed explicit representation of instruction-level parallelism is adequate to achieve the desired objectives, namely the ability of using the same object code for delivering increasing performance in wider implementations of a processor architecture, with minor degradation with respect to compiling/tuning the code for the specific implementations.

6. Concluding remarks

We have described an explicit representation of instruction-level parallelism suitable for wide-issue processors, in which programs are encoded as sequences of tree-instructions without reflecting the organization of the processor where they are executed. Tree-instructions are decomposed, during instruction cache reloading/accessing, into subtrees matching the resources available in an implementation. The decomposition is achieved without requiring complex hardware for dependency checking or dynamic scheduling of operations, and allowing simple instruction-dispatch logic.

Simulation results indicate that the representation of programs using tree-instructions is effective for executing the same statically-scheduled object code on implementations with varying issue-width, achieving ILP proportional to the issue-width of the implementations. Configurations with issue-width ranging from 6 to 16 were evaluated for a set of representative workloads, using the same object code in all cases; the ILP achieved in the smaller configurations exhibits minor degradation with respect to code compiled/tuned to those configurations (less than 10%), with some cases exhibiting better ILP.

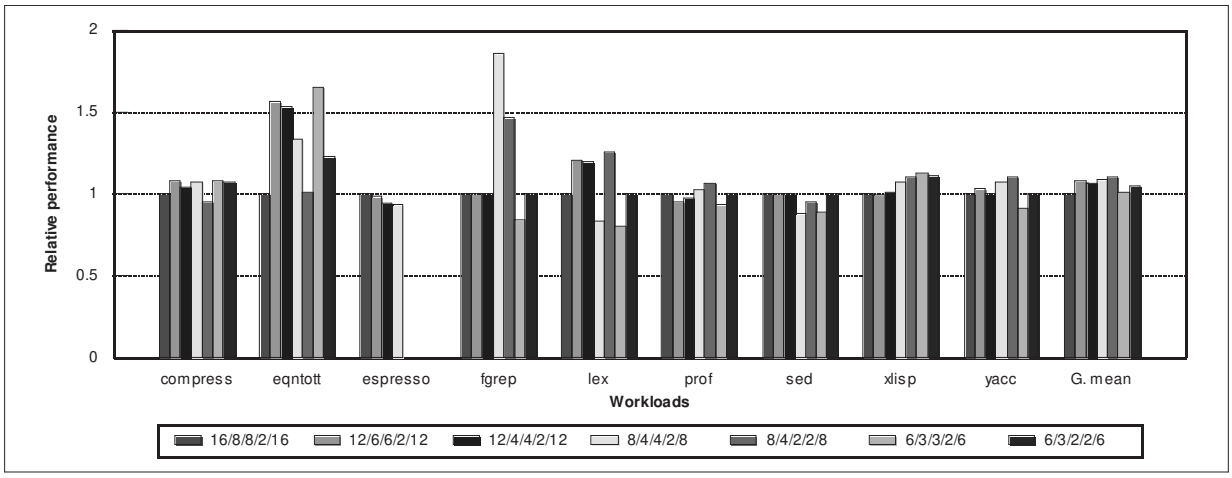

Figure 12: Relative performance of pruned code with respect to compiled code

In general, the ILP achieved with the approach proposed here varies monotonically with the issue-width of the implementations, whereas compiling/tuning the code to specific implementations seems to be more susceptible to variations in the behavior of the compiler optimization algorithms. Indeed, the compiler could produce code achieving equally good or better ILP in all cases, but that would require the development of a compiler optimized for every possible organization, and the generation of different code for each organization. In contrast, the pruning mechanism proposed here allows focusing the compiler development efforts in optimizing the code for the larger configurations, because the ILP achieved in the smaller ones will be adequate and proportional to the issue-width. Moreover, the proposed scheme makes possible using the same object-code in processors with widely different issue-width.

The results obtained with the approach described are due to the structure of the proposed tree-instructions. Pruning degrades ILP with respect to the execution of complete trees whenever the taken path is not the most probable one, and whenever the number of operations in the most-probable path exceeds the resources available in an implementation. On the other hand, as long as the taken path is fully contained within the first subtree, no ILP degradation occurs because the remaining subtrees are not executed.

A drawback of the explicit representation of ILP using the proposed approach is larger static code size with respect to compiling for a smaller implementation. Our experiments have shown an increase ranging from 10 to 20% throughout the different benchmarks. We plan to investigate the effects of this expansion on potential performance, focusing on the dynamic aspects.

Acknowledgments

We thank Rene Miranda for the development of the pruner and its integration into the simulation environment, and Kemal Ebcioglu for creating the conditions that made possible the development of these ideas. We also thank Brian Hall, Shy-Kwei Chen, Arkady Polyak, Norman Cohen, Richard Goldberg, Peter Oden, and Balaram Sinharoy for their contributions to different parts of CHAMELEON and the simulation environment.

References

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

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

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

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

[5] K. Ebcioglu, R. Groves, K.C. Kim, G. Silberman, I. Ziv, "VLIW compilation techniques in a superscalar environment," *ACM SIGPLAN Notices (PLDI/94),* Vol. 29, No. 6, pp. 36-48, June 1994.

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

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

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

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

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

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

[12] 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.

[13] B.R. Rau, "Dynamic scheduling techniques for VLIW processors," Technical Report HPL-93-52, Computer Research Center, Hewlett-Packard Company, June 1993.

[14] S. Melvin, M. Shebanow, and Y. Patt, "Hardware support for large atomic units in dynamically scheduled machines," in *Proceedings of 21st Annual International Symposium on Microarchitecture (MICRO-21)*, pp. 60-66, December 1988.

[15] M. Franklin and M. Smotherman, "A fill-unit approach to multiple-instruction issue," in *Proceedings of 27th Annual International Symposium on Microarchitecture (MICRO-27)*, pp. 162-171, December 1994.

[16] R. Nair, M. Hopkins, "Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups," Technical Report RC-20628, IBM T.J. Watson Research Center, November 1996.

[17] 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.

[18] 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.

[19] S.M. Moon, "Increasing instruction-level parallelism through multiway branching," in *International Conference on Parallel Processing,* Vol.2, pp.241-245, 1993.

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

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

[22] 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.

[23] M. Moudgill et al., "Compiler/architecture interaction in a tree-based VLIW processor," submitted to *Workshop on Compiler and Architecture Interaction, High-Performance Computer Architecture Conference,* 1997.

[24] 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.

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