# Execution-based Scheduling for VLIW Architectures

Kemal Ebcioğlu, Erik R. Altman, Sumedh Sathaye, Michael Gschwind
IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598
{kemal,erik,sathaye,mikeg}@watson.ibm.com

**Abstract.** We describe a new dynamic software scheduling technique for VLIW architectures, which compiles into VLIW code the program paths that are actually executed. Unlike trace processors, or *DIF*, the technique executes operations speculatively on multiple paths through the code, is resilient to branch mispredictions, and can achieve very large dynamic window sizes necessary for high ILP. Aggressive optimizations are applied to frequently executed portions of the code. Encouraging performance results were obtained on **SPECint95** and **TPC-C**. The technique can be used for binary translation for achieving architectural compatibility with an existing processor, or as a VLIW scheduling technique in its own right.
**Keywords:** Instruction-level parallelism, Dynamic compilation, Binary translation, Superscalar

## 1 Background and Motivation

VLIW architectures are desirable because they offer a simple hardware design path toward achieving wide issue at high frequency. However, architectural incompatibility with existing architectures, and hence the requirement to make software changes when migrating to a new VLIW architecture, has been a problem. In prior papers on the **DAISY** (**D**ynamically **A**rchitected **I**nstruction **S**et from **Y**orktown) project [1, 2], the authors have established techniques for achieving 100% architectural compatibility with an existing processor through software techniques applied to a wide issue VLIW. A number of difficulties were addressed, such as self modifying code, multi-processor consistency, memory mapped I/O, preserving precise exceptions while aggressively re-ordering VLIW code, and so on.

In the previous version of **DAISY**, the unit of translation was a page. Thus if execution reached a previously unseen page **P**, at address **X**, then all code on page **P** reachable from **X** — via paths entirely within page **P** — was translated to VLIW code. Any paths within page **P** that went offpage or that contained a register branch were terminated. At the termination point was placed a special type of branch that would (1) determine if a translation existed for the offpage/register location specified by the branch, and (2) branch to that translation if it existed, and otherwise branch to the translator. Once this translation was completed for address **X**, the newly translated code corresponding to the original code starting at **X** was executed.

In the previous **DAISY**, as well as in current version reported here, this translation/execution process begins at the bootstrap location for the emulated processor, for example, `0xFFF00100` for *PowerPC*. In this way, and as described in detail in [1, 2], the original base architecture can be properly emulated, with no need for any operating system or other changes.

We now define a few terms from our earlier work and make a few other observations in hopes of better illuminating what is new in our current work and what motivated it:

- As noted, the previous **DAISY** algorithm started at an entry point of a page, and scheduled along all paths reachable from that point. Because of the real-time constraints on the amount of time which may be spent scheduling operations, the scheduler did not build control-flow graphs and hence did not recognize join points — instead the code beyond join points was duplicated along however many paths pass through it. Because of this scheduling policy, the regions scheduled were trees which were in turn composed of tree VLIW instructions [3].
- We term these tree regions, **groups**. Likewise each leaf or exit of the tree, we term a **tip**. Since *groups* are trees, knowing by which *tip* the group exited, fully identifies the control path executed from the *group* entrance (or tree root).
- As befits **DAISY**'s real-time requirements, use of tree *groups* simplifies many areas of scheduling. For example, there is at most one reaching definition for each value. Trees also have drawbacks, in particular the duplicated code beyond join points can result in VLIW code that is many times larger than the code for the base architecture.
- The original **DAISY** had another big source of code explosion: since all paths from a given entry point were translated, *groups* could contain operations from paths which were rarely, if ever executed.

Thus, the previous **DAISY** technique was adequate for VLIW processors of modest width and large **ICaches**. However, for very wide machines, page crossings and indirect branches limited ILP. In this paper, we describe new techniques we have added to **DAISY** which overcome these ILP limitations and attack the code explosion problem. The present **DAISY**:

- Maintains the tree structure of groups, as well as tree instructions.
- Compiles only the executed portion of the code, by interpreting operations first. Thus **ICache** resources are spent more effectively.
- Crosses page boundaries and indirect branches, by making use of run time information to convert each indirect branch to a set of conditional branches. There is no limit on the number of pages a translated code fragment may cross. Only interrupts and code modification events are serializers.
- Conserves **ICache** and compile time resources by applying modest optimizations initially, and then scheduling aggressively with a large window size, only on the frequently executed portions of the code.

This **DAISY** approach can either be used as a binary translation system or as a VLIW scheduling technique in its own right. The rest of the paper is organized as follows. Section 2 describes the dynamic compilation algorithm. This algorithm includes not only the scheduling of operations, but rules for ending a scheduling region (Section 2.1), as well as a hardware/software mechanism for optimizing very frequently executed fragments of code (Section 2.2). Section 3 describes our performance evaluation experiments, Section 4 compares our approach to previous work, and Section 5 concludes.

## 2 The Dynamic Compilation Algorithm

In this section, we describe the execution based dynamic compilation algorithm. In what follows, the "*base architecture*" [4, 5] refers to the architecture with which we are trying to achieve compatibility, e.g., *PowerPC* or *S/390*. In this paper, our examples will be from *PowerPC*. To avoid confusion, we will refer to *PowerPC* instructions as *operations*, and reserve the term *instructions* for VLIW instructions (each potentially containing many *PowerPC operations*).

From the actually executed portions of the *base architecture* binary program, the dynamic compilation algorithm creates a VLIW program consisting of *tree regions*, which have a single entry (root of the tree) and one or more exits (terminal nodes of the tree).

The dynamic translation algorithm interprets code when a fragment of *base architecture* code is executed for the first time. As *base architecture* instructions are interpreted, the instructions are also converted to execution primitives (these are very simple RISC-style operations and conditional branches). These execution primitives are then scheduled and packed into VLIW tree regions which are saved in a memory area which is not visible to the *base architecture*. Any untaken branches, i.e., branches off the currently interpreted and translated trace, are translated into calls to the binary translator. Interpretation and translation stops when a stopping condition has been detected. (Stopping conditions are elaborated in section 2.1.) The last VLIW of an instruction group is ended by a branch to the next tree region.

Then, the next code fragment is interpreted and compiled into VLIWs, until a stopping condition is detected, and then next code fragment, and so on. If and when program decides to go back to the entry point of a code fragment for which VLIW code already exists, it branches to the already compiled VLIW code. Recompilation is not required in this case.

Looking at Figure 1(a), if the program originally took, path `A` through a given code fragment (where `cr1.gt` and `cr0.eq` are both false), and if the same path `A` through the code fragment (tree region) is followed during the second execution, the program executes at optimal speed within the code fragment — assuming a big enough VLIW and cache hits.

If at a later time, when the same tree region labeled `TR0` is executed again, the program takes a different path where `cr1.gt` is false, but `cr0.eq` is true (labeled path `B`), it branches to the translator, as seen in Figure 1(b). The translator may then start a new translation group at that point, or instead extend the existing tree region by interpreting *base architecture* operations along the second path `B` starting with the target of the conditional branch `if cr0.eq`. The *base architecture* operations are translated into primitives and scheduled into either the existing VLIWs of the region, or into newly created VLIWs appended to the region, as illustrated in Figure 1(c).

Assuming a VLIW with a sufficient number of functional units and cache hits, if the program takes path `A` or `B`, it will now execute at optimal speed within this tree region `TR0`, regardless of the path. This approach makes the executed code more resilient to performance degradation due to unpredictable branches.

The compilation of the tree region is necessarily never complete. It may have "loose ends" that may call the translator at any time. For instance, as seen in Figure 1(c), the first conditional branch `if cr1.gt` in tree region `TR0` is such

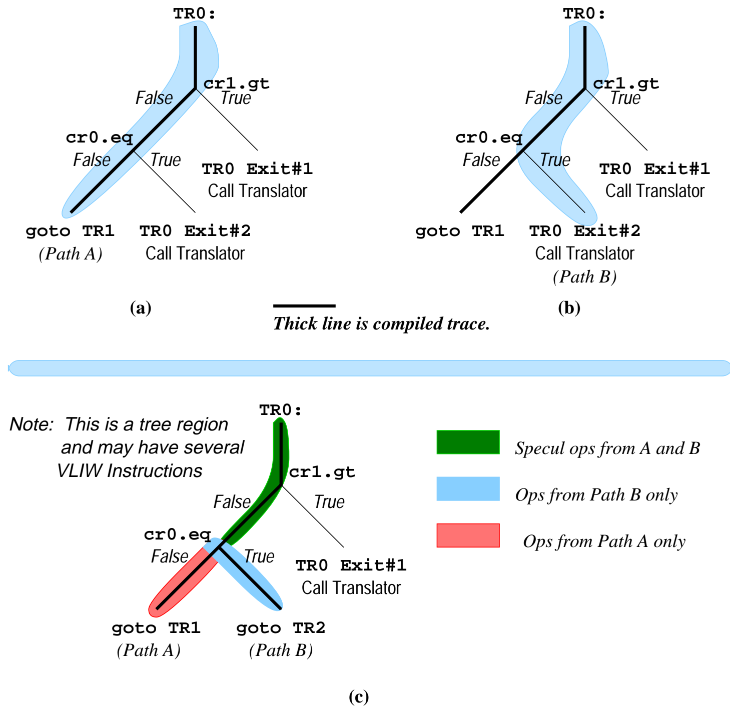


**Fig. 1.** Tree regions and where operations are scheduled from different paths.

a branch whose off-trace target is not compiled. Thus, dynamic compilation is potentially a never-ending task.

In our previous work, indirect branches always ended a tree region. This serialization is a significant impediment to high ILP, as such branches can occur every 25 branches or even more frequently in some programs. To avoid this problem, we note the address being branched to when an indirect branch is scheduled. For example the *PowerPC* `Link Reg` may contain `0x1234` on a `blr` instruction. Then, as in the example below, the `blr` can be converted from an indirect branch to a direct branch.

```
cmplr   cr8=r33,0x1234        # r33 holds PowerPC link register
if      (cr8.eq) goto L1234   # If lr==0x1234, goto translated code
                              # for PowerPC addr 0x1234
else    call_interpreter      # Start interpreting ops at addr in lr/r33
```

In this way, the operations found at `0x1234` can be scheduled into the current tree region. If other values of the `Link Reg` are encountered later in execution, explicit tests may be made for them as well.

### 2.1 Stopping Points for Paths in Tree Regions

Finding appropriate stopping points for a tree region is crucial for achieving high ILP, as well as for limiting the size of the generated VLIW code and translation time required for translation. Currently we consider ending a tree region at two types of operations:

- The **target** of a *backward branch*, typically a loop starting point, or
- A **subroutine entry** or **exit**, as detected heuristically through *PowerPC* `branch and link` or register-indirect branch operations.

Stopping (and hence starting) tree regions only at well-defined potential stopping points is useful, since if there was no constraint on where to stop, code fragments starting and ending at arbitrary *base architecture* operations could result, leading to unnecessary code duplication and increasing code expansion. Establishing well-defined starting points increases the probability of finding a group of compiled VLIW code when the translator completes translation of a tree region.

We emphasize that encountering one of the *stopping points* above does *not* automatically end a tree region. To actually end a tree region at a stopping point, at least one of the following *stopping conditions* must previously have been met:

- The desired ILP has been reached in scheduling operations, or
- The number of *PowerPC* operations on this path since the beginning of the tree region entry has exceeded a maximum *window size*.

The purpose of the ILP goal is to attain the maximum possible performance. The purpose of the *window size* limit is to limit code explosion — a high ILP goal may be attainable only by scheduling an excessive number of operations into a tree region.

### 2.2 Adaptive Scheduling Principles

In order to obtain the best performance, we do not make the ILP goal or maximum window size constants. Instead, a tree region is initially scheduled with modest ILP and window size parameters. If this region eventually executes only a few times, this represents a good choice for conserving code size and compile time.

If we later find that the time spent in a tree region tip is greater than a threshold fraction `thresh` of the total cycles spent in the program, then we optimize this area much more aggressively, e.g., using a much higher ILP goal and larger window size. Thus, if there are parts of the code which are executed more frequently than others (implying high re-use on these parts), they will be optimized very aggressively. If, on the other hand, the program profile is flat and many code fragments are executed with almost equal frequency, then no such optimizations occur, which could be good strategy for preserving **ICache** resources and translation time. Prior work on adaptive profiling-based optimizations includes the SUN *HotSpot* technology [6], for profile-guided optimization of frequently executed **JAVA** code fragments.

Determining whether a tree region tip is consuming a fraction greater than `thresh` of the total cycles can be done in a variety of ways. One possibility is to compile the profiling code into holes in the VLIW code. Another is to examine the current and previous program counter on timer interrupts enabled on transitions between tree regions.

In our results in Section 3, we employ a third way, namely an **8K** entry *8-way* set associative (hardware) array of cached counters indexed by the tip (exit point) of a tree region. These counters are automatically incremented upon exit from a tree region and can be inspected to see which tips are consuming the most time. They offer the additional advantages of not disrupting the **DCache** and being reasonably accurate.

## 3 Performance Evaluation

VLIW projects usually involve compilers and simulators to run the compiled code, they do not use traces as inputs. In this round of experiments, we have used a trace based evaluation methodology, which gives us access to kernel as well as application traces. Here, we report results for **SPECint95** and **TPC-C**. Our **SPECint95** traces are from **RS/6000** *PowerPC* machines. Each trace consists of **50**, *2 million* operation samples, uniformly sampled over a run of the benchmark. The **TPC-C** trace is slightly longer, but similarly obtained.

The performance evaluation tools implement the dynamic compilation strategy using a number of tools:

- A tree-region former reads a *PowerPC* operation trace and forms tree-regions according to the strategy described in this paper. However, to avoid translating short-lived groups (tree regions), groups are interpreted 30 times before translation. Also, a group is allowed to extend into a true multiple path tree only if a premature exit from the group is executed frequently. The initial region formation parameters were: **ILP goal=3**, *window size limit=24* operations. When **5%** of the time is spent on a given tree region tip, the tip is aggressively extended with **ILP goal=10**, *window size limit=180*. An **8K** entry, **8-way** associative array of counters were simulated, to detect the frequently executed tree region tips, as described in Section 2.2.
- A VLIW scheduler schedules the *PowerPC* operations in each tree region and generates VLIW code according to the clustering, functional unit and register constraints, and determines the cycles taken by each tree region tip.
- A VLIW instruction memory layout tool lays out VLIWs in memory according to architecture requirements.
- A multi-level **ICache** simulator determines the **ICache** CPI penalty using a history-based prefetch mechanism.
- A multi-level **DCache** and **DTLB** simulator. The data references in the original trace are run through these simulators for hit/miss simulation. To account for the effects of speculation and joint cache effects on the off chip **L3**, we multiplied the **DTLB** and **DCache CPI** penalties by a factor of *1.7* when calculating the final **CPI**. We chose *1.7* based on speculation penalties we have previously observed in an execution-based model. To account for disruptions due to execution of translator code, we flush on-chip caches periodically based on a statistical model of translation events.

From the number of VLIWs on the path from the root of a tree region to a tip, and the number of times the tip is executed, we can calculate the total number of VLIW cycles. Empty VLIWs are inserted for long latency operations, so each VLIW takes one cycle. The total number of VLIWs executed, divided by the original number of *PowerPC* operations in the trace, yields the infinite cache, but finite resource **CPI**.

Stall cycles due to caches and TLBs, are tabulated using a simple *stall-on-miss* model for each cache or TLB miss. In the *stall-on-miss* model everything in the processor stops when a cache miss occurs, or data from a prior prefetch is not yet available.

To model **translation overhead**, we first define **re-use rate**:

$$\textbf{Re-use Rate} = \frac{\text{Number of Dynamic Ins in Trace}}{\text{Number of Unique Ins Addresses in Trace}}$$

**Reuse rates** are shown in the last column of Table 1, and are in millions. **SPECint95 rates** were measured through an interpreter based on the reference inputs although operations in library routines were not counted [1]. The **TPC-C** value was obtained from the number of code page faults in a benchmark run. **Re-use rates** may be used to estimate **translation overhead** (in terms of CPI) as follows:

- $\#P$ = # of Times an Operation undergoes *Primary* Translation
- $\#S$ = # of Times an Operation undergoes *Secondary* Translation
- $CP$ = Cycles per *Primary* Translation of an Operation
- $CS$ = Cycles per *Secondary* Translation of an Operation

Then

$$\textbf{Overhead} = \frac{\#P \times CP + \#S \times CS}{\textbf{Re-use Rate}}$$

The translation (or *primary* translation) of a *PowerPC* operation occurs when it is being added to a tree region for the first time. A *secondary* translation of an operation occurs when it is already in a tree region while new operations are being added to the tree region. In this study we have used an estimate of 4000 cycles for a *primary* translation and 800 cycles for a secondary translation. Our **DAISY** experience yielded about 4000 PowerPC operations to translate one *PowerPC* operation [2]. Secondary translation merely requires disassembling VLIW code and reassembling it, something we estimate to take about 800 cycles.

| Program | Inf Resrc CPI | Resrc CPI Adder | Inf Cache CPI | *CPI Adders* ICache | DCache | TLB | Xlate Overhd (CPI) | **Final CPI** | Avg Win-dow | Code Explo | *Reuse Rate* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **li** | 0.37 | *0.00* | 0.37 | *0.00* | 0.01 | *0.00* | 0.00 | **0.38** | 0.8 | 21.7 | *16.5* |
| **m88k** | 0.19 | *0.08* | 0.28 | *0.00* | 0.00 | *0.00* | 0.00 | **0.29** | 0.7 | 36.7 | *14.8* |
| **ijpeg** | 0.18 | *0.13* | 0.31 | *0.00* | 0.01 | *0.00* | 0.00 | **0.33** | 1.2 | 50.2 | *10.3* |
| **vortex** | 0.22 | *0.06* | 0.28 | *0.00* | 0.13 | *0.02* | 0.00 | **0.44** | 1.0 | 41.1 | *3.4* |
| **perl** | 0.30 | *0.08* | 0.38 | *0.00* | 0.00 | *0.00* | 0.00 | **0.38** | 1.1 | 36.4 | *6.8* |
| **compr** | 0.39 | *-0.01* | 0.38 | *0.00* | 0.14 | *0.01* | 0.00 | **0.52** | 0.7 | 26.8 | *69.2* |
| **go** | 0.60 | *-0.04* | 0.56 | *0.04* | 0.06 | *0.00* | 0.00 | **0.67** | 7.2 | 16.0 | *6.2* |
| **gcc** | 0.38 | *0.02* | 0.41 | *0.03* | 0.01 | *0.00* | 0.01 | **0.46** | 2.7 | 20.2 | *0.74* |
| **GMean** | | | 0.36 | | | | | **0.42** | 1.8 | 30.8 | *8.1* |
| **TPC-C** | 0.29 | *0.09* | 0.39 | *0.03* | 0.20 | *0.03* | 0.00 | **0.65** | 0.8 | 27.7 | *3.8* |

**Table 1.** Performance on **SPECint95** and **TPC-C**.

Infinite cache CPI with the approach described here is roughly 30%-40% better than the old page-based **DAISY**. Table 1 details the performance of our current approach on a 16 issue machine configuration where the 16 total

[1] We are indebted to Jay Leblanc for providing us this data.

| Cache | Size | Linesize | Assoc | Latency |
|---|---|---|---|---|
| **L1–I** | *64K* | 1K | **8** | 1 |
| **L2–I** | *1M* | 2K | **8** | 3 |
| **L1–D** | *32K* | 256 | **4** | 2 |
| **L2–D** | *512K* | 256 | **8** | 4 |
| **L3** | *32M* | 256 | **8** | 42 |
| **Memory** | – | – | – | 150 |
| **DTLB1** | *128 Entries* | – | **2** | 2 |
| **DTLB2** | *1K Entries* | – | **8** | 4 |
| **DTLB3** | *8K Entries* | – | **8** | 10 |
| **Page Table** | – | – | – | 90 |

**Table 2.** Cache and TLB Parameters.

operations can include up to 8 **Load/Stores**. The machine is divided into 4 clusters of 4 functional units each, for high frequency operation. Within a cluster back to back dependent operations are allowed, but when a cluster is crossed an extra cycle is incurred. **L1 DCache**s are duplicated in each cluster, but stores are broadcast to all copies. Cache and TLB parameters are given in Table 2. The configurations used are quite aggressive, to tolerate speculation and the large code explosion that results from the present approach.

The *Infinite Resource* CPI column of Table 1 describes the CPI of a machine with infinite registers and resources, constrained only by serializations between tree regions, and realistic operation latencies (including a load latency of 3 cycles for unsigned loads and 4 for algebraic loads). The *Finite Resource CPI Adder* describes the extra CPI due to finite registers and function units, as well as clustering effects, and possibly compiler immaturities. (This value can sometimes be negative, since the load latency for the finite resource ILP measurement is 2 cycles). *Infinite Cache CPI* is the sum of the first two columns. The **ICache, DCache** and **DTLB CPI** describe the additional **CPI** incurred due to **ICache**, **DCache**, and **TLB** misses, assuming the *stall-on-miss* machine model described above. *Translation Overhead* is determined using the formulas and values above. *Final CPI* is then the sum of the *Infinite Cache CPI*, **ICache**, **DCache**, **TLB**, and *Overhead* columns. The initial interpretation overhead is insignificant. Also, there are no branch stalls, due to our zero-cycle branching technique [3, 7].

Even though unlike previous infinite cache VLIW studies our model takes into account all the major CPI components, we have not modeled the VLIW machine at a very detailed (*e.g., RTL*) level. Hence performance could fall short of the numbers presented here. However, our model also omits some potential performance enhancers, such as software value prediction, software pipelining, tree-height reduction, and **DCache** latency tolerance techniques.

Note that the VLIW **ICache** consists of 8 independent mini-**ICache**s corresponding to $\frac{1}{8}$th of a VLIW supplying a pair of ALUs. Thus the **L2** mini-**ICache** size is logically 128K instead of 1M, and the mini-**ICache** linesize is 256 bytes instead of 2K bytes. But because each such mini-**ICache** has to have a redundant copy of the branch fields to reduce the wire delays, the physical size is larger than the logical size. The VLSI technology and packaging needed for this design will probably be realizable on a single chip within a few years.

The *Average Window Size* in Table 1 indicates the average dynamic number of *PowerPC* operations between tree region crossings. The *Code Explosion* indicates the ratio of translated VLIW code pages to *PowerPC* code pages. Our mean code explosion of 1.8 is more than 2× better than the old page-based **DAISY**. This improvement has come about largely because of our use of adaptive scheduling techniques and the fact that only executed code is translated.

Preliminary experiments on large multi-user systems indicate that a translation space of **2K-4K** *PowerPC* pages is sufficient to cover the working set for code. With a code explosion factor of 1.8×, such large multi-user systems would likely require **15 − 30 Mbytes** for VLIW code:

$$(2K/4K) \text{ pages} \times 4K \text{ bytes per page} \times 1.8 \text{ Code Explosion} \approx 15M/30M$$

**15 − 30 Mbytes** for translated code will probably be affordable on moderate size systems over the next few years.

## 4 Related Work

Previous work in inter-system binary translation has largely focused on easing migration between platforms. To this end, problem state executables were translated from a legacy instruction set architecture to a new architecture. By restricting the problem domain to a single process, a number of simplifying assumptions can be made about execution behavior and the memory map of a process. Dynamic binary translation of programs as a translation strategy is exemplified by caching emulators such as **FX!32** [8]. **FX!32** emulates only the user program space and depends on support from the OS (*Microsoft* **Windows NT**) to provide a native interface identical to that of the original migrant system.

The presented approach is more comparable to full system emulation, which has been used for performance analysis (e.g., **SimOS** [9]) and for migration from other legacy platforms as exemplified by **Virtual PC**, **SoftPC/SoftWindows** and to a lesser extent **WABI**, which intercepts **Windows** calls and executes them natively. Full system simulators execute as user processes on top of another operating system, using special device drivers for *virtualized* software devices. This is fundamentally different from our approach which uses dynamic binary translation to implement a processor architecture. Any operating system running on the emulated architecture can be booted using our approach.

The present approach is different from the **DIF** approach of Nair and Hopkins [10]. It schedules operations on multiple paths to avoid serializing due to mispredicted branches. Also, in the present approach, there is virtually no limit to the length of a path within a tree region or the ILP achieved. In **DIF**, the length of a (single-path) region is limited by machine design constraints (e.g., 4-8 VLIWs). Our approach follows an all software approach as opposed to **DIF** which uses a hardware translator. This all-software technique allows aggressive software optimizations hard to do by hardware alone. Also, the **DIF** approach involves almost three machines: the sequential engine, the translator, and the VLIW engine. In our approach there is only a relatively simple VLIW machine.

*Trace processors* [11] are similar to **DIF** except that the machine is out-of-order as opposed to a VLIW. This has the advantage that different trace fragments do not need to serialize between transitions between one trace cache entry and another. However, when the program takes a path other than what was recorded in the trace cache, a serialization can occur. The present approach solves this problem by incorporating an arbitrary number of paths in a software trace cache entry, and by very efficient zero overhead multiway branching hardware [7].

The dynamic window size (trace length) achieved by the present approach can be significantly larger than that of trace processors, which should allow better exploitation of ILP.

## 5 Conclusion

We have described the latest version of **DAISY**, which employs a dynamic software translation approach whereby operations from the actual execution path of a base architecture such as *PowerPC* are scheduled into VLIW instructions. The proposed technique allows operations from multiple code pages and can schedule operations through indirect branches. This technique can also schedule operations from multiple paths. The proposed technique is adaptive, and schedules more aggressively on frequently executed paths. This technique exposes significant ILP, with values reaching almost 2.5 instructions per cycle even after acounting for cache effects.

## References


1. K. Ebcioğlu and E. Altman. **DAISY:** Dynamic Compilation for 100% Architectural Compatibility. Research Report RC 20538, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1996.
2. K. Ebcioğlu and E. Altman. **DAISY:** Dynamic Compilation for 100% Architectural Compatibility. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pages 26–37, Denver, CO, June 1997. ACM.
3. K. Ebcioğlu. Some Design Ideas for a VLIW Architecture for Sequential-Natured Software. In M. Cosnard et al., editor, *Parallel Processing*, pages 3–21. North-Holland, 1988. (Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing).
4. G. M. Silberman and K. Ebcioğlu. An Architectural Framework for Migration from CISC to Higher Performance Platforms. In *Proc of the 1992 International Conference on Supercomputing*, pages 198–215, Washington, DC, July 1992. ACM Press.
5. G. M. Silberman and K. Ebcioğlu. An Architectural Framework for Supporting Heterogeneous Instruction-Set Architectures. *IEEE Computer*, 26(6):39–56, June 1993.
6. Sun Microsystems. The Java Hotspot Performance Engine Architecture. http://java.sun.com/products/hotspot/whitepaper.html, April 1999.
7. K. Ebcioğlu, J. Fritts, S. Kosonocky, M. Gschwind, E. Altman, K. Kailas, and T. Bright. An eight-issue tree-VLIW processor for dynamic binary translation. In *Proc. of the 1998 International Conference on Computer Design (ICCD '98) – VLSI in Computers and Processors*, pages 488–495, Austin, TX, October 1998. IEEE Computer Society.
8. A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates. FX!32–A Profile-Directed Binary Translator. *IEEE Micro*, 18(2):56–64, March 1998.
9. M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete Computer Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*, 3(4):34–43, Winter 1995.
10. R. Nair and M. Hopkins. Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups. In *Proc of the 24th Annual International Symposium on Computer Architecture*, pages 13–25, Denver, CO, June 1997. ACM.
11. E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace Processors. In *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 138–148, Research Triangle Park, NC, December 1997. IEEE Computer Society.


This article was processed using the LaTeX macro package with LLNCS style