VHC: Quickly Building an Optimizer for Complex Embedded Architectures

Michael Dupré, Nathalie Drach and Olivier Temam
LRI, Paris South University, France
{dupre,drach,temam}@lri.fr

Abstract

To meet the high demand for powerful embedded processors, VLIW architectures are increasingly complex (e.g., multiple clusters), and moreover, they now run increasingly sophisticated control-intensive applications. As a result, developing architecture-specific compiler optimizations is becoming both increasingly critical and complex, while time-to-market constraints remain very tight.

In this article, we present a novel program optimization approach, called the Virtual Hardware Compiler (VHC), that can perform as well as static compiler optimizations, but which requires far less compiler development effort, even for complex VLIW architectures and complex target applications. The principle is to augment the target processor simulator with superscalar-like features, observe how the target program is dynamically optimized during execution, and deduce an optimized binary for the static VLIW architecture. Developing an architecture-specific optimizer then amounts to modifying the processor simulator which is very fast compared to adapting static compiler optimizations to an architecture. We also show that a VHC-optimized binary trained on a number of data sets performs as well as a statically-optimized binary on other test data sets. The only drawback of the approach is a largely increased compilation time, which is often acceptable for embedded applications and devices. Using the Texas Instruments C62 VLIW processor and the associated compiler, we experimentally show that this approach performs as well as static compiler optimizations for a much lower research and development effort. Using a single-core C60 and a dual-core clustered C62 processors, we also show that the same approach can be used for efficiently retargeting binary programs within a family of processors.

1. Introduction and Related Work

In this article, we present a novel approach for deriving a scheduler for a complex architecture in very little time and effort. The main drawback of the technique is that compilation time is much longer than in traditional compilers. However, in embedded systems, compilation is usually an off-line process performed before the system is delivered to the end-user, so that long compilation times can be acceptable. And in any case, even if the VHC optimizer is later replaced with a traditional compiler, the VHC approach can significantly improve time-to-market by quickly providing a scheduler based on the existing architecture simulator. Moreover, the approach can be also applied to high-performance computers and any performance-intensive program which is not frequently modified/recompiled.

VLIW architectures rely on efficient compiler technology to extract ILP and schedule instructions. Currently, most compiler optimization techniques rely on a *machine model* embedded in the compiler. However, as hardware complexity increases (e.g., cluster-related issues [29], variable latencies, irregular VLIW instruction encoding constraints [31] or operand path constraints) embedding an accurate and reliable machine model in a compiler is increasingly difficult, which, in turn, limits the efficiency of purely static optimization techniques. That embedded applications are no longer restricted to simple DSP-like kernels but are inching toward high-level programmed applications with pointer analysis issues [22], complex control-flow and complex memory behaviors further complicates the static optimization approach.

For that reason, dynamic or dynamically-assisted compiler optimizations have received increased attention in the past few years. The principle is to use run-time information to fine-tune static program transformation parameters (e.g., unrolling factor, tile size) and thus to compensate for the lack of accuracy of the embedded machine model. While program performance is improved, the overall compiler development effort is about the same (or sometimes even greater) than for purely static compiler optimization techniques. Moreover, most of the research work on profile-based and feedback-directed compilation [8, 9, 27, 15] has focused on demonstrating the potential performance of the approach, rather than on finding practical ways to implement them. The difficulty is to show that it is possible and how to aggregate program behavior statistics over several runs in order to derive a program version that can perform well over a range of target data sets [14]. For that reason, profile-based and feedback-directed optimization techniques are not much used in practice, even though they have

Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004)
0-7695-2102-9/04 $20.00 © 2004 IEEE

been implemented in commercial compilers [4, 5]. Iterative compilation [11, 37] is a similar approach which highlights the potential benefits and limitations of dynamic optimization techniques: based on an exhaustive or smart search of the static transformation parameter space, it can find very efficient program versions but requires many runs of the same program on the same data set; it is both more efficient and even less practical to use. A more practical way to use dynamic information is dynamic compilation, as proposed in DAISY [18], FX!32 [13], Dynamo [10], Mojo [12] and more recently in DELI [17]. The principle is to collect run-time information but to use it immediately to modify the program binary. Also, Vachharajani et al. [37] have recently proposed a method intermediate between iterative and dynamic compilation which consists in deriving at compile-time several program versions and pruning versions at run-time. Overall, these techniques are more practical and can be as efficient as profiled-based compiler optimizations, but they require a rather complex run-time environment to support them; in embedded systems where memory resources are scarce and cost constraints are strict, this environment may become an excessive overhead.

Almost all the above techniques consist in using traditional static compiler optimizations (at the source or binary level) and feeding them with run-time information in one way or another. In other words, the compiler development effort for all these techniques remains quite high. To alleviate this difficulty, several intermediate approaches propose to combine program transformations with hardware-based mechanisms to (often speculatively) optimize programs at run-time without incurring the hardware cost of out-of-order superscalar processors. Code Morphing in the Crusoe [16] speculatively optimizes translated x86 binary code, DISVLIW processors [26] dynamically schedule and group instructions within VLIW words at run-time by taking into account run-time dependencies, and Region Slips [35] use the trace-based rePlay [32] framework to build and cache static speculative schedules at run-time. Unlike software dynamic optimization techniques, these combined hardware/software techniques have access to the processor inner workings, i.e., the available dynamic information is not limited to the narrow interface provided by hardware counters or other dedicated sampling mechanism. And more important, part of the transformation effort is given to the architecture, much like in superscalar processors.

In this article, we present a novel approach, called the *Virtual Hardware Compiler* (VHC), for quickly optimizing program schedules on statically controlled VLIW-like architectures. The principle is to find an efficient instruction schedule by running the binary code on a cycle-level simulator of the target VLIW architecture *augmented* with superscalar-like dynamic reordering features, and monitoring the schedules induced by the dynamic scheduling mechanism of this modified architecture. The different observed dynamic schedules are combined to find the most appropriate static schedule, and a new static binary code is generated. Even though the Virtual Hardware Compiler is a purely software approach, it does *not* rely on static program transformations driven by dynamic information as in profile-based, feedback-directed or dynamic compilation techniques. On the contrary, it works almost exactly like the hardware scheduling techniques used in out-of-order superscalar processors, so that it dynamically/implicitly adjusts to even complex architectures, unlike static or profile-based optimizations such as Trace Scheduling [21]. But unlike combined hardware/software or pure hardware optimization techniques, it requires absolutely no hardware support. The VHC approach bears some similarity with the Software Trace Cache proposed by Ramirez et al. [33] in the sense that it emulates a hardware optimization using a purely software approach. Unlike many profile-based and feedback-directed compilation techniques, the VHC approach is designed to easily aggregate the statistics of several runs. And we experimentally show that VHC-optimized programs perform as well as statically-optimized programs in average over several test data sets, distinct from the VHC training data sets.

This technique has three major assets: (1) the main one is that building an optimizing compiler (the scheduling part) for a target architecture takes no more time than writing the dynamically scheduled version of the statically scheduled processor simulator which, for the TI C62, took about 30 man-days; (2) the schedule simultaneously takes into account several dynamic phenomena like cache misses or conditional branch behavior which are typically difficult to harness using static optimizations [23]; while embedded processors currently have few such dynamic features, they may become widespread in high-performance embedded processors as performance requirements increase, e.g., cache and branch predictions in the Intel XScale processor [3]; (3) we find that a VHC-optimized binary performs similarly, in average, than a statically optimized binary. In addition, the VHC can be used to improve design-space exploration, since it can provide an estimate of program performance on an intermediate design as if a target optimizer were already available. Finally, small application-specific processor vendors who cannot always afford significant optimizer efforts might be interested in a low-cost optimizer approach based on their simulators.

Since we make almost no assumption on the program or the architecture, we can apply this technique to a wide range of applications, including control-intensive applications, and a large range of architectures, from complex VLIW architectures with heterogeneous and irregular features to in-order superscalar architectures. Using the Texas Instruments C62 VLIW processor, we have experimentally shown that this Virtual Hardware Compiler can perform as well as static optimizations on SpecInt programs without the hassle of developing an architecture-specific backend static optimizer. Moreover, when the Virtual Hardware

Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004)
0-7695-2102-9/04 $20.00 © 2004 IEEE

Compiler is applied on top of classic static optimizations, it can further improve program performance by 11% in average on the TI C62 by taking into account dynamic behavior such as cache misses or conditional branches, without the need for sophisticated static analysis as with profiled-based techniques. Using the C60 and C62 processors, we also show how the Virtual Hardware Compiler approach can also serve as a retargetable framework for a family of VLIW processors based on the same ISA. Retargeting then amounts to augmenting the simulator of the new processor then plugging it into our environment, i.e., the retargeting effort is of the order of *user retargetability* in Leupers's classification [30].

The article is organized as follows: in Section 2, we present the principles of the Virtual Hardware Compilation mechanism, in Section 3 the experimental framework, and in Section 4, we provide experimental evidence of the performance of our approach by comparing the Virtual Hardware Compiler performance on two TI processors against the TI compiler; we have also applied the VHC to another processor platform (the Alpha 21164 in-order superscalar processor) and another compiler platform (a combination of the SUIF [38] front-end and the SPAM [7] back-end).

2. The Virtual Hardware Compiler

Building a virtual dynamically scheduled processor. In short, we want our statically scheduled processor to behave like an out-of-order superscalar processor. For that purpose, we need to add to the architecture (in fact, to the architecture simulator) a number of features usually found in superscalar processors. However, a major difference with real architecture design is that we do not have the constraint of truly implementing a dynamically scheduled processor architecture. On the contrary, we need to add these dynamic features while retaining all the latencies and architecture specifics of the original statically scheduled processor. Therefore, we can exploit this relative freedom to implement a more simple dynamic reordering mechanism than the one found in real architectures.

Consider a standard VLIW architecture in Figure 1(a); the corresponding "augmented" dynamically scheduled architecture is shown in Figure 1(b). The main new components are: an instruction window, a set of reservation stations and register renaming logic. Unlike in a true superscalar architecture, we can release a number of constraints: a large and fully-associative instruction window, a large number of fully-associative reservation stations, no output bus conflict, branch prediction is perfect, and there is no need for a reorder buffer (ROB) to implement in-order commits since true dependencies (registers and memory) are naturally preserved during code generation. On the other hand, unlike in a true superscalar processor the number of physical registers is necessarily the same as the number of logical registers. However, in order to accommodate various program transformations, we do need to have a register renaming mechanism. Moreover, unlike in some superscalar processor implementations, register renaming cannot be *implicit*, i.e., handled by the architecture using reservation stations or a ROB, it must be *explicit* and reflected in the newly generated binary. Finally, we need to preserve a number of constraints which reflect the original architecture such as functional units latencies, cache or other memory data structure latencies, original pipeline structure, and the maximum number of register reads and writes per cycle.

Generating a new binary based on dynamic scheduling. Once the virtual dynamically scheduled processor simulator is available, we can use it as a software scheduler. In a first step, the original binary is executed. During execution, we monitor which instructions are executed in parallel: more precisely, which instructions are issued to the functional units in the same cycle. Each such set of instructions is termed an *instruction pattern* and recorded. Progressively, we build a *dictionary* of all patterns and their frequency of occurrence. Consider the example in Figure 2; Figure 2(a) shows the operations control flow (each operation *instance* is identified by a unique identifier I_k, much like a PC), and Figure 2(b) how operations are grouped in VLIW instructions and statically scheduled by a traditional compiler. On the virtual superscalar version of the VLIW processor, operations are independently and dynamically scheduled, resulting in multiple different schedules during execution, as shown in Figure 2(c), e.g., pattern `I2//I5` does not correspond to one of the initial VLIW words.

After execution, on a second step, we use the collected dictionary to generate a new binary. First, we sort basic blocks by their frequency of occurrence. We then pick the most frequently occurring sequence of consecutively executing basic blocks (1, 2 or 3 in our experiments) and want to find a *coverage* of this basic block sequence using the instruction patterns in the dictionary, i.e., intuitively rewrite the basic blocks as suggested by dynamic scheduling. For that purpose, we sort instruction patterns that only contain instructions of these basic blocks by both their frequency of occurrence and size (number of instructions). Then, we consider the first pattern in the two sorted lists: if they match, we select it as it is both large (it exploits ILP) and frequently occurring; then we consider the first 2 patterns in each list and select any as yet unselected intersection pattern; then, we consider the first 3 patterns and so on until all the instructions in the basic blocks are covered by one of the instruction patterns. The selected instruction patterns are the VLIW words of the new optimized binary code. In the example of Figure 2, `BB1`→`BB2` is the most frequently executed path. The dictionary of all patterns observed during execution that only contain instructions of these two basic blocks is listed in Figure 2(d) by frequency of occurrence and size. The coverage algorithm starts, the first item in each list are distinct, but the first two items have a non-

Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004)
0-7695-2102-9/04 $20.00 © 2004 IEEE

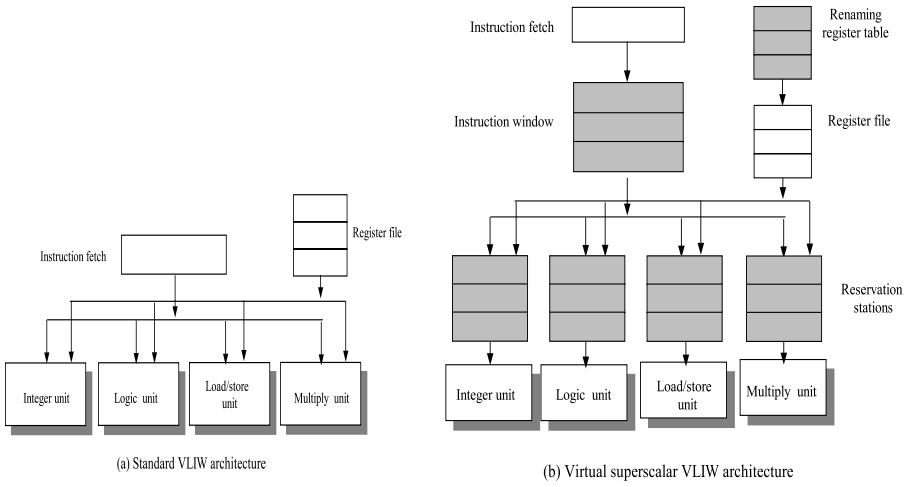


Figure 1. *Original and "superscalarized" VLIW architectures*

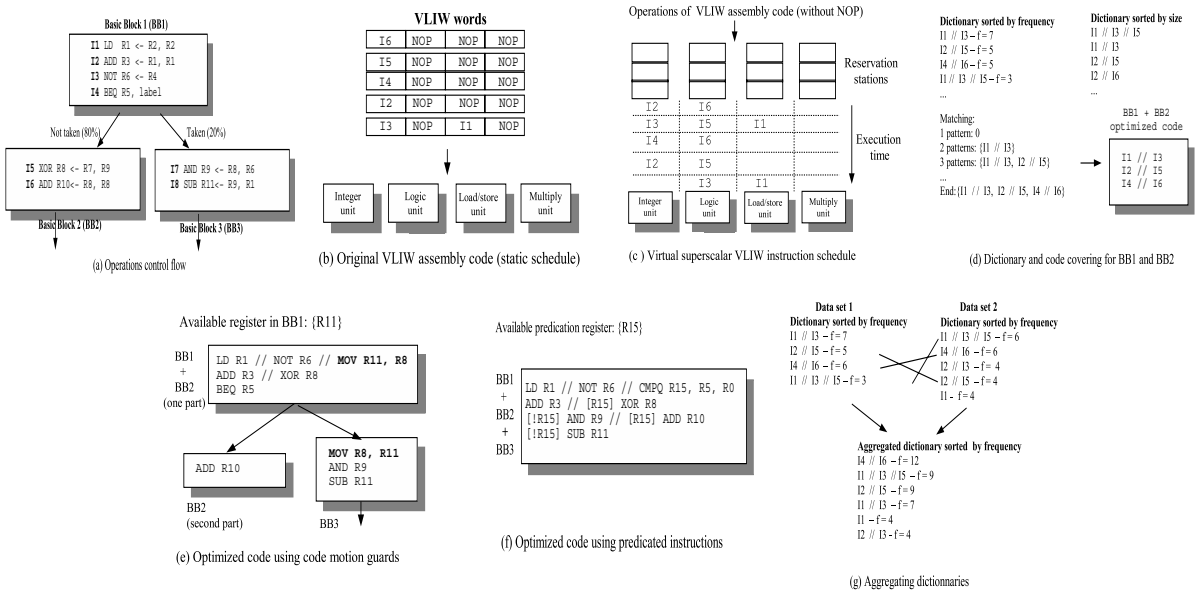


Figure 2. *Example of Virtual Hardware Compilation*

empty intersection {I1//I3}, which is selected, then the scope of the match is progressively extended until all operations are covered. For the sake of completion, all single operations are added at the bottom of the lists. The resulting optimized code is shown in Figure 2(d).

Since dynamic reordering can move instructions across basic block boundaries, the VHC must either support predicated instructions if available (see Figure 2(f)), or add recovery code, as for operation I5 in Figure 2(e), which is moved above conditional branch I4, as in trace or superblock scheduling [21, 25]. In either case, the VHC needs to have registers for predication or backing up values. Since the VHC optimizes at the binary level, registers have already been allocated, so it must find "locally available" registers. For that purpose, we use a simple heuristic: a register is considered "locally available" in a sequence of instructions starting with instruction I_s and ending with instruction I_e if the register is not used in any instruction of the sequence, and if it is written the first time it is used in all the possible paths of the control flow graph following I_e. Consider again the example of Figure 2(f), and let us first assume the processor supports predication. Then, we need to find an available register to act as predication register for operation I5. In our example, register R11 is neither read nor

Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004)
0-7695-2102-9/04 $20.00 © 2004 IEEE

written in BB2 (we also assume it is written the first time it is used after BB2) and it is written in BB3, so it is locally available and can serve as a predication register (assuming predication registers are general-purpose registers as for the Texas Instruments C62), see Figure 2(f). Let us now assume the processor does not support predication. If we execute I5 and branch I4 is taken, the value of R8 is incorrect, so we need to backup the value of R8, and R11 can serve as a backup register. We insert the code motion guard MOV R11, R8 in BB1, and the corresponding recovery code MOV R8, R11 in BB3 in case the executed path is BB1,BB3 instead of the scheduled target path BB1,BB2 (see Figure 2(e)). In this step, we verify that true dependencies are preserved and we add NOPs to build VLIW words and to hide instruction latencies.

Load/Store alias analysis. The binary-level load/store alias analysis proposed by Fernandez et al. [20] has been partially implemented in the Virtual Hardware Compiler. They have demonstrated that it is possible to perform efficient speculative alias analysis on binary programs, sometimes at the cost of inserting recovery code. Only the *safe* version of their analysis is currently implemented in the VHC because the *unsafe* version requires a larger amount of recovery code and thus free registers. The safe method computes a symbolic expression of the load and store addresses and determines if these expressions are equivalent if possible. The unsafe method is based on profiling and consists in checking that two pointers address apparently non-overlapping memory regions; because the unsafe method is speculative, it requires the addition of recovery code.

Aggregating the statistics of several data sets. In order to build a binary that would perform well over a range of data sets, we want to somehow aggregate the information gathered over several runs. We will later discuss in Section 4 on the generality and performance stability of this binary; in this paragraph, we are solely concerned with the method for aggregating the information.

In the VHC, aggregating information is particularly easy, because the way we build the code is already based on the aggregated information of several executions of the same basic blocks. Since these executions may indifferently belong to the same run (same data set) or several runs (several data sets), to aggregate the information of several runs, we only need to merge the statistics of the dictionary of each run. Thus, if the same pattern is found in several dictionaries, we can combine the statistics of the different dictionaries. To combine dictionaries, we simply add up the pattern frequencies from the different dictionaries. Figure 2(g) shows the dictionaries of two data sets, and the corresponding aggregated dictionary. For instance, pattern I1//I3//I5 has a frequency of 3 in the dictionary of data set 1, 6 in the dictionary of data set 2, and thus 9 in the aggregated dictionary.

Correctness of the generated program. Program correctness is achieved by construction, the true dependencies are preserved, system calls and exception handling code are not optimized. Naturally, an instruction pattern may not be valid over all executions (e.g., an instruction in the target block of a branch has been moved above the branch which is not taken). It is the role of the recovery code to ensure correctness. Recovery is possible because whenever an instruction that is moved modifies a register, this register is backed up. The local availability of registers determines how much recovery code can be added. When it is not possible to add a recovery code because registers are not available, the program is simply not modified.

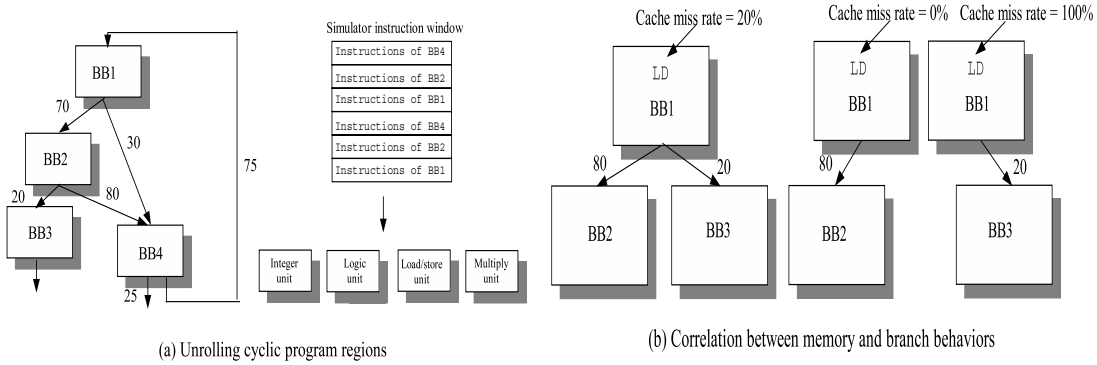
Also, the dictionary may not always contain patterns that only contain instructions of the 1, 2 or 3 considered basic blocks. In that case, we simply do not optimize the corresponding block. We are currently exploring sliding basic block windows to increase the optimizing opportunities.

Optimizations implicitly performed by the Virtual Hardware Compiler. Not surprisingly, an analysis of the optimized binaries shows that the VHC optimizations are fairly similar to some static compile-time optimizations: the VHC implicitly unrolls loops with the appropriate unrolling factor and moves instructions to hide long latencies, and it exploits ILP across basic blocks by speculating on the most probable branch path. While the main assets of the VHC over static optimizations is fast and easy development and automatic adjustment to complex architecture behavior, the VHC still has unique optimization capabilities that either give it a performance edge or compensate for the lack of more sophisticated static optimization techniques.

(a) The VHC can unroll program regions that do not necessarily correspond to true source-code loops; any program region where the program stays long enough can be viewed as a "local" loop and is implicitly unrolled, however complex its control flow structure. Consider for instance the set of basic blocks in Figure 3(a), and assume the program frequently iterates over the sequence BB1,BB2, BB4. In a superscalar processor or the VHC, the instruction window will contain multiple consecutive instances of this sequence, in effect unrolling the sequence and implicitly interpreting it temporarily and locally as a loop. Consequently, the VHC can hide latency and exploit ILP in program regions that evade static analysis, which is especially useful in complex control-intensive programs. Still Lavery and Hwu [28] showed that they can apply modulo scheduling to some complex control-flow regions.

(b) The VHC implicitly exploits correlation among multiple architecture characteristics (cache misses, conditional branch behaviors, ...). Doing the same with profile-based optimizations can be tedious: not only simultaneously exploiting multiple profiling information in static program transformations can be fairly complex, but significant architecture-specific profiling research work is also needed to understand the correlation among profiling statistics. Consider the example of Figure 3(b): the LD in-

Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004)
0-7695-2102-9/04 $20.00 © 2004 IEEE



(a) Unrolling cyclic program regions

(b) Correlation between memory and branch behaviors

Figure 3. *Implicit Virtual Hardware Compiler optimizations*

struction in basic block `BB1` has 20% misses; in 80% of the cases, `BB1` branches to `BB2` so a profile-based optimizer and the VHC will choose to optimize for the `BB1,BB2` path. If the memory latency is high, e.g., 40 cycles, then Amdahl's Law suggests to schedule the `BB1`→`BB2` sequence assuming the `LD` misses, and that is what a profile-based optimizer relying on cache miss statistics would do. However, if it appears that the misses only occur in the 20% cases where `BB1` branches to `BB3`, the best option is to schedule `BB1`→`BB2` assuming `LD` hits. In the VHC, it will implicitly happen: the schedule corresponding to the case where `LD` misses and the sequence `BB1`→`BB2` is executed will never occur, the VHC dictionary will only contain patterns corresponding to two correlation cases: `LD` hits and the sequence `BB1`→`BB2` is executed, or, `LD` misses and the sequence `BB1`→`BB3` is executed

3. Experimental Framework

Architectures, compilers and simulators. Most of the experiments are based on the Texas Instruments C6x family of embedded VLIW processors, and more specifically the C60 (also called Omap) and the C62 (TMS320C6000 DSPs family [6]). The C60 is a VLIW processor that executes 4 operations/instructions per cycle (4 functional units); it has 16 general-purpose registers, two of which can serve as predicating registers. The C62, see Figure 4, is a clustered version of the C60 with two cores linked by a crossbar for accessing the remote register. A core can read up to two remote registers per cycle and only arithmetic/logic functional units can perform remote reads. Unlike in other versions like the C64, forwarding of remotely read operands is not enabled in the C62 which further constrains static instruction ordering.

For the two target processors (C60 and C62), we used the C62x simulator developed by Cuppu [36] from which we have derived a C60 simulator, and the Texas Instruments Code Composer Environment 2.2 (the Texas Instruments compiler). We used 150-million instruction traces to build

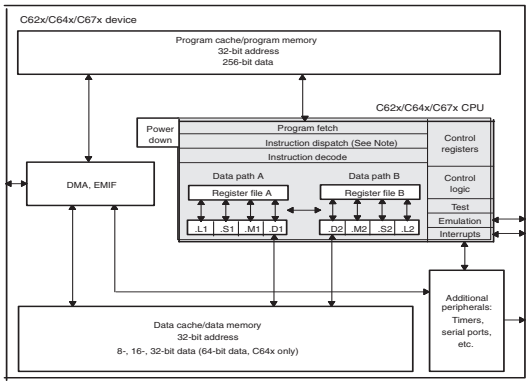


Figure 4. *The C62x architecture*

the dictionaries; the trace starts when the first instruction of the most time-consuming procedure is executed for the first time. In order to execute the Spec benchmarks on the TI, we use the standard C libraries provided by TI in their Code Composer Environment; some system calls had to be modified to enable correct execution. Non-optimized binaries are obtained with -o0, -mu (disables software pipelining), and -g/-s (disables optimizations between C statements like constant propagation); note that register allocation, dead code elimination and interblock dependency analysis are still active. Optimized binaries are obtained with -o3 (includes unrolling and software pipelining) and -pm (program-level optimizations). We measured that the execution time speedup of optimized binaries over non-optimized binaries is 1.13 in average on the Alpha using the HP `cc` compiler; these results are consistent with the official Spec statistics [2].

To build the out-of-order versions of the TI and Alpha processors for the VHC, the TI and Alpha simulators were augmented with the following dynamic reordering features (see Figure 1): a 5000-entry fully-associative instruction window, 100-entry fully-associative reservation stations per functional unit, perfect branch prediction.

To validate and compare the VHC against another compiler platform, we used a combination of two publicly avail-

Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004)
0-7695-2102-9/04 $20.00 © 2004 IEEE

Benchmark	Data sets
`Gzip`	`Ref, Train, Test`: Spec data set
	`Iso`: filesystem file
	`Bin`: /usr/bin from Linux system
`Parser`	`Ref, Train, Test`: Spec data set
	`Gcc doc`: gccc compiler documentation
	`Web page`: slashdot mainpage
`Eon`	`Ref, Train, Test`: Spec data set
	`Sphere`: 150x150 pixels sphere
	`Table`: 300x300 pixels table in a corner
`Crafty`	`Ref, Train, Test`: Spec data set
	`Chess1`: layout with full board (depth going from 1 to max/2)
	`Chess2`: layout with half the whites and full blacks (depth going from 1 to max)
`Twolf`	`Ref, Train, Test`: Spec data set
	`HP`: layout taken from the MCNC project
	`Xerox`: layout taken from the MCNC project
`FIR`	`Ref`: TI sample - 100x100 image
	`Image1`: 320x200 black and white image
	`Image2`: 320x200 color image
`FFT`	`Ref`: Ref: TI sample - 500 random values
	`Set1`: 500 complex random values
	`Set2`: 1024 complex random values
`IDCT`	`Ref`: TI sample - 50x50 matrix
	`Set1`: 100x100 matrix of random simple precision values
	`Set2`: 100x100 matrix of random simple precision values
`Gouraud`	`Ref`: TI sample - 3d object
	`Sphere`: 320x200 pixels with 1 light source to shade on 1 texture
	`Cube`: 320x200 pixels scene with 1 light source to shade on 3 textures

Table 1. *Benchmarks and data sets*

able compilers: SUIF [38] for the front-end combined with SPAM [7] for the back-end targeting to the C62. Finally, we also applied the VHC to the Alpha 21164, an in-order superscalar processor, using the SimAlpha [1] simulator and the Alpha commercial compiler.

Benchmarks and data sets. The benchmarks are the subset of the control-intensive SpecInt 2000 suite (`Crafty, Eon, Gzip, Parser, Twolf`) that we could compile and run successfully on all platforms. We also used a second set of more simple DSP-like benchmarks from Texas Instruments: `FIR, FFT, IDCT and Gouraud`. The `AVG` bar on all graphs corresponds to the average performance over all benchmarks.

With respect to data sets, to our knowledge, there is no widespread benchmark suite with multiple data sets per benchmark. Therefore, we elaborated on the SpecInt suite, and besides the `test, train, ref` data sets, we built several additional data sets for each benchmark, see Table 1, and we built a similar data set suite for the TI benchmarks; for instance, for `gzip` we used Linux kernel binaries and an `iso` file segment as additional data sets. To have reasonably different data sets, we profiled the programs with each data set and checked that the IPC of the most frequently used procedures differ by 10% or more from one data set to all others. In the future, we will explore whether Eeckout et al. [19] method for analyzing data set distance is useful for the VHC experiments.

For each benchmark and for each data set, we have derived an optimized binary using the VHC. In the different experiments we always test the VHC on a given data set after having trained it with the remaining data sets. The test data set is *never* one of the training data sets unless otherwise explicitly specified. For each graph and each experiment within a graph, the VHC performance for a given benchmark is the average performance over all possible combinations of training+test data sets; the number of possible combinations is equal to the number of data sets (a combination is $n-1$ training data sets and 1 test data set, where n is the number of data sets for the benchmark: from 3 to 5).

4. Performance Analysis

In this section, we first provide general performance results for the VHC, then analyze various parameters and applications of the VHC.

A VHC-optimized binary is no less general than a statically optimized binary. Figure 5 shows the VHC performance for each train/test data set combination, i.e., unlike in the other graphs of this section, the VHC performance is not averaged over all data set combinations (the terms "train data set" and "test data set" should not be confused with `Train` and `Test` data sets of the Spec2000 benchmarks). We find that the average performance of a VHC-optimized binary is usually similar or higher than the performance of a statically optimized binary. Moreover, we find that the performance variance of the VHC and the static optimizer are fairly close, which means that the performance stability of the VHC across a range of data set is about the same as that of the static optimizer.

These experiments contradict the common wisdom that program optimizations based on dynamic data perform well only on very similar data sets and that the resulting program is less general than a statically optimized program. Looking at the workings of many static optimizations, this result is not all that surprising. In fact, static optimizations set values such as load latencies or branch outcomes either arbitrarily or using static analysis based on the architecture model embedded in the compiler. For instance, when a load is scheduled as if it would hit in cache, static optimizations *speculate* on the load behavior; the same for branch outcomes and other architecture or program-related instructions behaviors. In the VHC these values are similarly set, except that they are set *based on dynamic data*; while this data will vary from one data set to another, many loads (and many branches,...) will have a similar behavior across data sets so that the VHC choices are likely to be more informed than the speculative static compiler choices in many cases, especially if the architecture is complex and the architecture model embedded in the compiler is not accurate enough.

Now, this discussion has important practical consequences: there is no reason why a VHC-optimized binary cannot be used for a large range of data sets just like

Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004)
0-7695-2102-9/04 $20.00 © 2004 IEEE

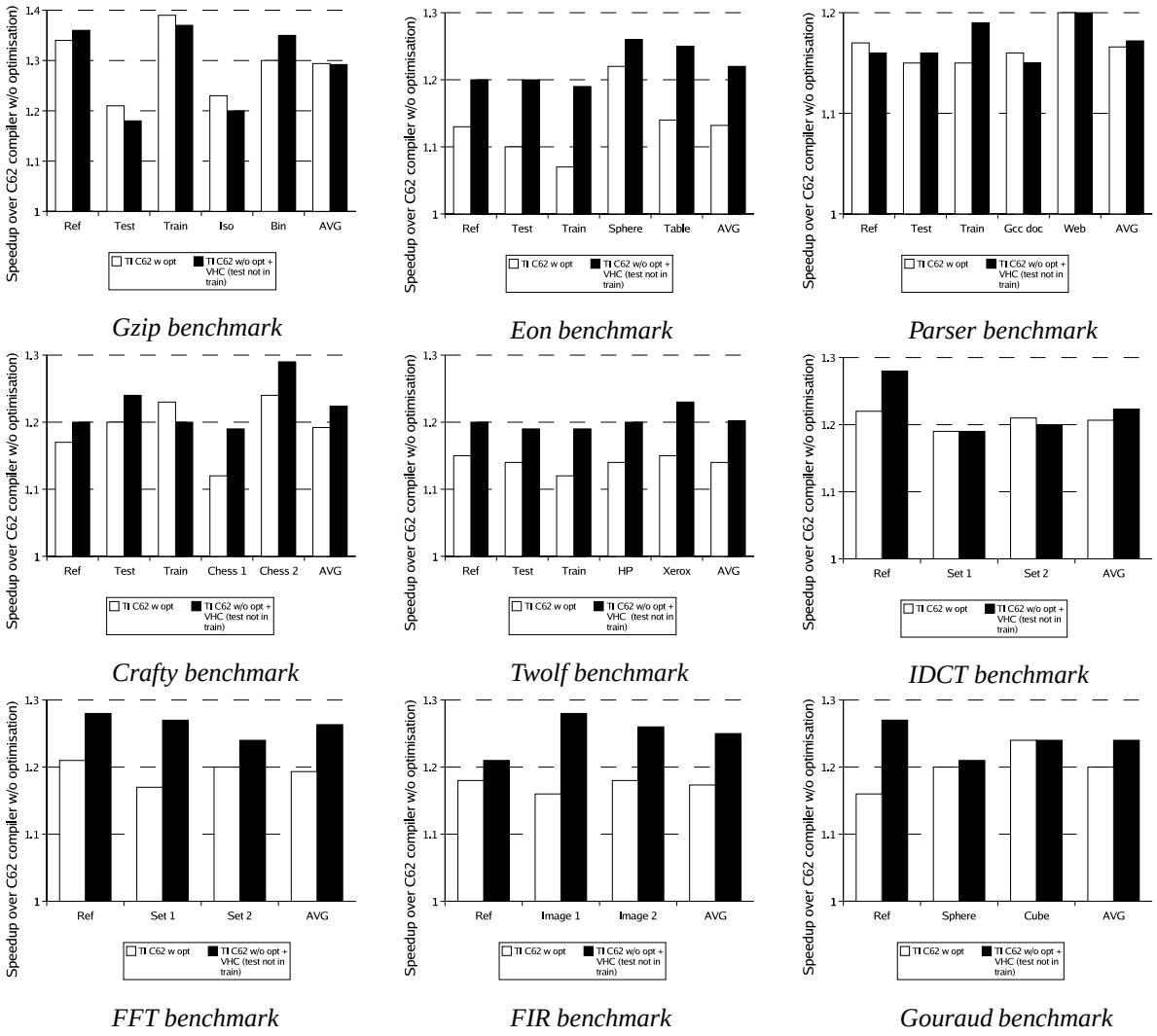


Figure 5. *VHC performance across data sets*

a statically optimized binary, as demonstrated by the experiments of Figure 5. In other words, the VHC can be used as a true optimizing compiler and can be just as reliable performance-wise as a static optimizing compiler.

VHC versus static optimizations+profiling. The TI compiler can perform profile-based optimizations; unlike the VHC, it cannot aggregate profiles across multiple data sets, so the train data set is the test data set. Therefore, some of the experiments of Figure 6 are biased in favor of the TI compiler. We provide three sets of experiments with the VHC: the case where the train data sets is just the test data set as for the TI compiler, the case where all data sets are used for training including the test data set, and the case where all data sets are used for training except the test data set, as in all the other experiments of this section. We find that even in the latter case which is the least favorable for the VHC, the VHC outperforms the TI compiler with profiling. In static compilers, profiling information is fed to one or a few static optimizations in a pre-determined manner, while in the VHC, the dynamic behavior of the architecture can affect the whole instruction schedule, and implicitly, dynamic information is exploited in a broader way.

Impact of block lookup and number of registers. Figure 7 shows the VHC performance for intra-block and inter-block (2 and 3) optimization, see Section 2. Intra-block performs relatively poorly as expected, because there are few optimizing opportunities; on the other hand, inter-block optimization improves program performance by 27% and more in average over the non-optimized versions, and it outperforms the TI static compiler optimizations (often with 2 blocks, always with 3 blocks).

However, increasing the number of target basic blocks beyond 3 did not bring significant performance improvements because too few registers are available to insert additional recovery code. More generally, the number of registers as well as register availability in target basic blocks

Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004)
0-7695-2102-9/04 $20.00 © 2004 IEEE

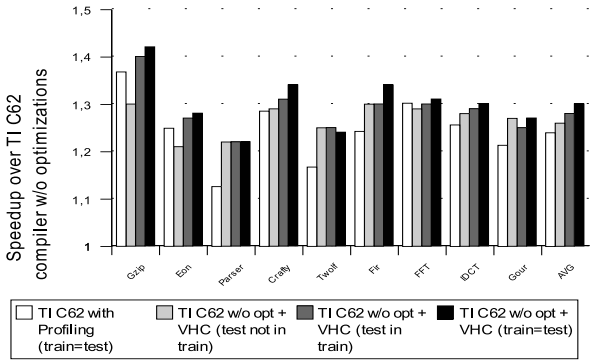


Figure 6. *Virtual Hardware Compiler versus profile-based static optimizations*

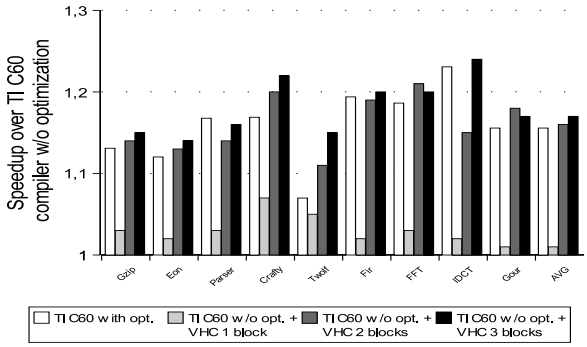


Figure 8. *Illustrating the register constraint with the C60*

strongly determine VHC performance. For instance, the VHC achieves slightly lower speedups on a C60 Omap processor because this processor has only 16 registers compared to 32 in the C62, see Figure 8. Unless otherwise specified, we use inter-block optimizations with 3 blocks throughout the experiments of the section.

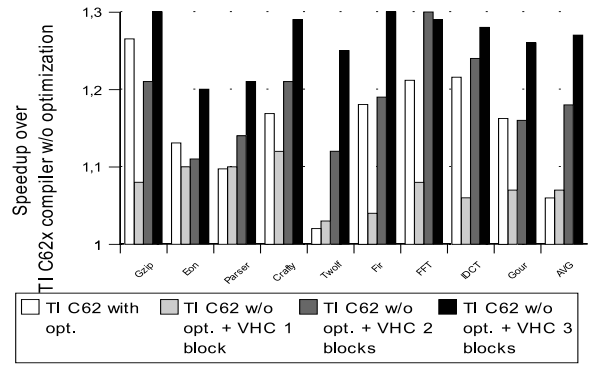


Figure 7. *Virtual Hardware Compiler versus static optimizations on the C62; inter-block optimizations over 1, 2 or 3 blocks*

Reducing optimized code size.

Recovery code overhead. As explained in Section 2, the Virtual Hardware Compiler must insert recovery code in order to perform inter-block optimizations, and as a result, optimized code size increases. If a given set of basic blocks is scarcely executed, adding recovery code will not increase performance but it always increases code size. Therefore, we want to preferentially apply inter-block optimizations to the most frequently executed basic blocks, and limit useless code size increase. For that purpose, we define a *recovery code overhead threshold* which is the maximum code size increase tolerated. Whenever this threshold is reached, the optimization process stops. Figures 9 and 10 show the performance and code size increase of the Virtual Hardware Compiler for different threshold values. Usually performance increases with the threshold size because more basic blocks are optimized. However, the more basic blocks are optimized, the more likely some of the branch path predictions will prove wrong, and the more overhead code is executed (in the wrong paths), the lesser the performance. Consequently, even in terms of performance only, there is an optimum value for the overhead threshold, see Figure 9, graphs `TI C62 w/o opt. + VHC C62 10, 20, 30%`.

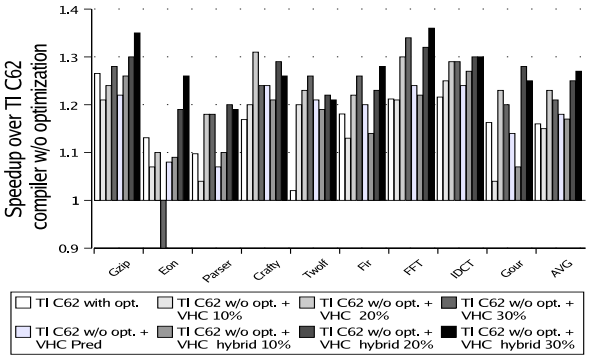


Figure 9. *Comparing different recovery code techniques (speedup)*

Predicated instructions. Using predicated instructions instead of recovery code can reduce the amount of overhead code but it can also increase the number of executed instructions and thus degrade performance, see Figures 9 and 10. The TI processor has predicated instructions but experi-

Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004)
0-7695-2102-9/04 $20.00 © 2004 IEEE

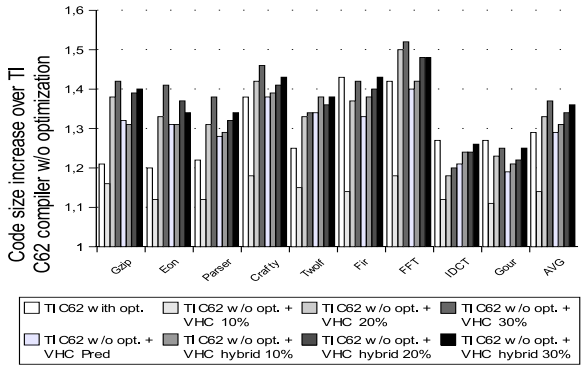


Figure 10. *Comparing different recovery code techniques (code size increase)*

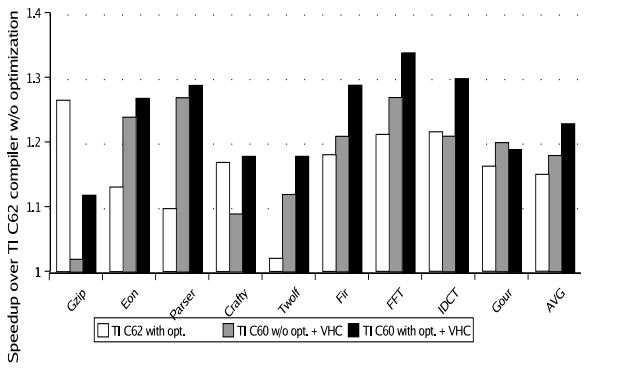


Figure 11. *The Virtual Hardware Compiler as a retargeting tool*

ments show that the predicated version of the optimized binary generated by the VHC performs no better than the version with recovery code. The predicated version is not quite as effective as expected on the TI processor because this processor has only 2 general-purpose registers per cluster available for predication. As a result, only few instructions can be predicated which, in turn, limits the length of the basic block sequence that can be reordered. For instance, with `parser`, only 2 blocks can be reordered in average (7% of the code) versus 3 blocks with recovery code (15% of the code).

A hybrid technique. While the VHC approach calls for additional registers for predication or recovery code, we naturally cannot expect such architecture modifications in the short term. As a result, we investigated a hybrid approach for architectures with a limited number of registers (general-purpose or predication registers). This approach combines both predication and control code to schedule with the lowest possible code size increase. Predication is applied first whenever possible, and recovery code is inserted only if no more predication register is available. Figures 9 and 10 show that this approach yields good performance and reduced code size increases for almost all benchmarks compared with the recovery code approach, see graphs `TI C62 w/o opt. + VHC C62 hybrid 10,20,30%`. In some cases like `gzip` the hybrid technique usually outperforms the two other techniques. In other cases like `IDCT`, the VHC even generates a smaller code than the TI compiler because some static optimization techniques (unrolling, software pipelining,...) can significantly increase code size. In all other experiments, the `VHC` bars refer to the hybrid technique; for the training data set, we used the optimum threshold, and then for the test data set, a fixed threshold corresponding to the average optimum threshold over all the trained data sets.

Retargeting the Virtual Hardware Compiler. In order to illustrate the ability of the VHC to rapidly adjust to architecture modifications, we have considered two different versions of the C6x processor: the C62 dual-core processor used in previous sections with a crossbar between both cores, and a simpler version, the C60 Opale, a single-core version of the C62 processor without any crossbar. While both processors come with their own compiler, the main difference between both tools is essentially crossbar-oriented optimizations in the C62 compiler.

The experiment consists in generating a binary for the C60 processor using its dedicated compiler, and then optimizing this binary for the C62 dual-core processor using the VHC only, i.e., without using the C62 compiler. Implicitly, we want to show that the VHC can be quickly retargeted as a processor family evolves, that it can adjust to complex architecture modifications and still generate efficient code. Figure 11 shows that whether the VHC is applied to non-optimized or optimized C60 binaries, in average, it performs slightly better than the C62 compiler. Since the C60 and C62 processor simulators were fairly similar, "superscalarizing" the C62 after the C60 required only 14 man-days, i.e., retargeting the VHC to the C62 was very fast.

The Virtual Hardware Compiler as a complement to traditional static compilers. As mentioned above, the ability to rapidly adjust to complex architectures is the strong asset of the VHC approach. On the other hand, static compilers sometimes perform complex program transformations, such as loop fusion, that are beyond the capabilities of the VHC. Consequently, both techniques are sometimes complementary. In Figure 12, we have applied the VHC on top of their respective static compilers for the TI C60, TI C62 and Alpha processors, and compared the static compiler performance against the static compiler combined with the VHC. As expected, programs with complex dynamic behaviors like the SpecInt benchmarks benefit most from the additional optimizations, improving performance by 10% in average for TI C62 and Alpha.

Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004)
0-7695-2102-9/04 $20.00 © 2004 IEEE

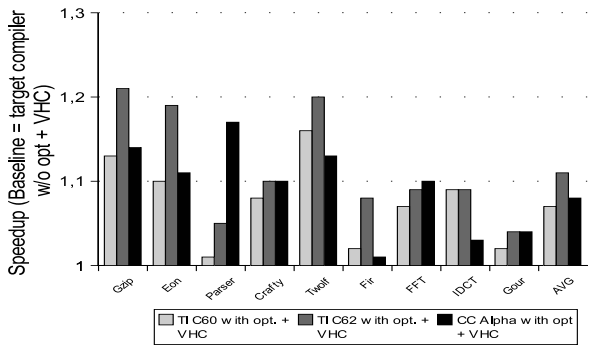


Figure 12. *Virtual Hardware Compiler combined with static optimizations*

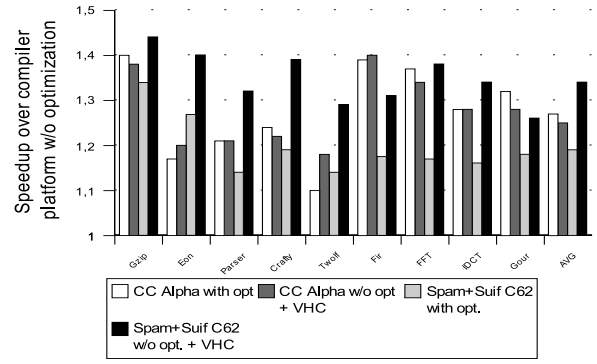


Figure 13. *Virtual Hardware Compiler on other architecture and compiler platforms.*

Other architecture and compiler platforms. In order to evaluate the VHC approach on more than one architecture and compiler, we have built a compiler platform using a combination of SUIF [38] for the front-end and SPAM [7] for the back-end (SPAM is a retargetable open-source back-end for which a TI C62 version was already available), and we have also applied the VHC to the Alpha using SimAlpha as the base simulator and the HP `cc` compiler. Figure 13 confirms that the performance improvements and variations are fairly similar with these platforms and with the TI.

Optimization time. Naturally, optimization time is the main drawback of the VHC approach. Since optimizing actually means simulating a program on one or preferably multiple data sets using a cycle-level simulator, optimization time evaluates in hours versus seconds for a traditional compiler. VHC compilation includes two phases: the most time-consuming phase is the simulation phase which runs at a few hundred thousand instructions per second (typical cycle-level simulator speed), and it is followed by a coverage and binary transformation phase.

We currently use 150-million instruction traces to build the dictionaries, and the trace starts when the first instruction of the most time-consuming procedure is executed for the first time. In the future, we will explore the impact of the SimPoint technique proposed by Sherwood et al. [34] who have shown that it is possible to extract representative program behaviors from a 100-million instruction trace provided the trace starting point is carefully chosen. Such a technique can potentially improve the dictionary accuracy. In addition, DiST [24] can further speedup simulation by a factor of 20 or more at a loss of accuracy less than 5% by distributing simulation over several machines. Combining these different techniques to speedup Virtual Hardware Compilation could progressively widen its scope beyond embedded applications and general-purpose applications where compilation time is not critical.

5. Conclusions

In this article, we have presented the Virtual Hardware Compiler, a technique for quickly optimizing a program for statically controlled VLIW-like architectures, which consists in augmenting the target processor simulator with superscalar-like features. Using the TI C62 processor, we have shown that this technique can perform as well or better than static compiler optimizations. The main asset of the VHC is the small effort required to build a targeted optimizer, compared to a classic static optimizer.

References

[1] http://www.simplescalar.com/v4test.html.

[2] http://www.spec.org/cpu2000/results/cpu2000.html.

[3] Intel xscale microarchitecture. *http://developer.intel.com/design/intelxscale/*.

[4] *Compaq C Compiler User's Guide, http://www.hp.com*, 2000.

[5] *TMS320C6000 Optimizing Compiler User's Guide (Rev. K) http://dspvillage.ti.com*, 2002.

[6] Tms320c6000 cpu and instruction set reference guide. *Texas Instruments Literature, Number SPRU189C*, March, 1999.

[7] Spam research group. *SPAM Compiler User's Manual, http://www.ee.princeton.edu/ spam/*, September, 1997.

[8] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *Conference on Programming Language Design and Implementation*, pages 85–96, 1997.

[9] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone. *Symposium of Operating Systems Principles*, pages 1–14, 1997.

Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004)
0-7695-2102-9/04 $20.00 © 2004 IEEE

[10] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[11] F. Bodin, T. Kisuk, P. M. W. Knijnenburg, M. F. P. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. *Workshop on Profile and Feedback Directed Compilation, in conjuction with the International Conference on Parallel Architectures and Compilation Techniques*, 1998.

[12] W.-K. Chen, S. Lerner, R. Chaiken, and D. Gillies. Mojo: A dynamic optimization system. *Workshop on Feedback-Directed and Dynamic Optimization, in conjunction with the International Symposium on Microarchitecture*, 2000.

[13] A. Chernoff, M. Herdeg, and R. Hookway. Fx!32 a profile-directed binary translator. *International Symposium on Microarchitecture*, pages 56–64, 1998.

[14] R. Cohn and P. G. Lowney. Design and analysis of profile-based optimization in compaq's compilation tools for alpha. *Journal of Instruction-Level Parallelism*, April, 2000.

[15] T. M. Conte, M. N. Kishore, and M. A. Hirsch. Accurate and practical profile-driven compilation using the profile buffer. *International Symposium on Microarchitecture*, pages 36–45, 1996.

[16] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. *International Symposium on Code Generation and Optimization*, pages 15–24, 2003.

[17] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. Deli: A new run-time control point. *International Symposium on Microarchitecture*, pages 257 – 268, 2002.

[18] K. Ebcioglu and E. R. Altman. Daisy: Dynamic compilation for 100% architectural compatibility. *International Symposium on Computer Architecture*, pages 26–37, 1997.

[19] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Workload design: Selecting representative program-input pairs. *International Conference on Parallel Architectures and Compilation Techniques*, pages 83–94, 2002.

[20] M. Fernandez and R. Espasa. Speculative alias analysis for executable code. *International Conference on Parallel Architectures and Compilation Techniques*, pages 222–231, 2002.

[21] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers, C-30(7):478-490*, 1981.

[22] R. Ghiya, D. M. Lavery, and D. C. Sehr. On the importance of points-to analysis and other memory disambiguation methods for c programs. *Conference on Programming Language Design and Implementation*, pages 47–58, 2001.

[23] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. *International Conference on Supercomputing*, pages 317–324, 1997.

[24] S. Girbal, G. Mouchard, A. Cohen, and O. Temam. Dist: A simple, reliable and scalable method to significantly reduce processor architecture simulation time. *International Conference on Measurement and Modeling of Computer Systems*, pages 1–12, 2003.

[25] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: an effective technique for vliw and superscalar compilation. *Journal of Supercomputing*, pages 229–248, 1993.

[26] S. Jee and K. Palaniappan. Dynamically scheduling vliw instructions with dependency information. *Workshop on Interaction between Compilers and Computer Architectures, in conjunction with International Symposium on High-Performance Computer Architecture*, pages 15–26, 2002.

[27] C. Krintz. Coupling on-line and off-line profile information to improve program performance. *International Symposium on Code Generation and Optimization*, pages 69–78, 2003.

[28] D. M. Lavery and W. W. Hwu. Modulo scheduling of loops in control-intensive non-numeric programs. *International Symposium on Microarchitecture*, pages 126–137, 1996.

[29] R. Leupers. Instruction scheduling for clustered vliw dsps. *International Conference on Parallel Architectures and Compilation Techniques*, pages 291–300, 2000.

[30] R. Leupers. Compiler design issues for embedded processors. *IEEE Design and Test of Computers*, pages 51–58, July/August, 2002.

[31] J. Liu, T. Kong, and F. Chow. Effective compilation support for variable instruction set architecture. *International Conference on Parallel Architectures and Compilation Techniques*, pages 56–67, 2002.

[32] S. Patel and S. Lumetta. replay : a hardware framework for dynamic program optimization. *IEEE Transactions on Computers, 50(6):300-318*, June, 2001.

[33] A. Ramirez, J.-L. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero. Software trace cache. *International Conference on Supercomputing*, pages 119–126, 1999.

[34] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, 2002.

[35] F. Spadini, B. Fahs, S. Patel, and S. S. Lumetta. Improving quasi-dynamic schedules through region slip. *International Symposium on Code Generation and Optimization*, pages 149–158, 2003.

[36] S. Srinivasan, V. Cuppu, and B. L. Jacob. Transparent data-memory organizations for digital signal processors. *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 44–48, 2001.

[37] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-space exploration. *International Symposium on Code Generation and Optimization*, pages 204– 215, 2003.

[38] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J.-A. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. Suif: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12):31–37, 1994.

Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004)
0-7695-2102-9/04 $20.00 © 2004 IEEE