

# From Sequences of Dependent Instructions to *Functions*: An Approach for Improving Performance without ILP or Speculation

Sami Yehia and Olivier Temam  
LRI, Paris XI University, France  
{yehia,temam}@lri.fr

## Abstract

*In this article, we present an approach for improving the performance of sequences of dependent instructions. We observe that many sequences of instructions can be interpreted as functions. Unlike sequences of instructions, functions can be translated into very fast but exponentially costly two-level combinational circuits. We present an approach that exploits this principle, speeds up programs thanks to circuit-level parallelism/redundancy, but avoids the exponential costs.*

*We analyze the potential of this approach, and then we propose an implementation that consists of a superscalar processor with a large specific functional unit associated with specific back-end transformations. The performance of the SpecInt2000 benchmarks and selected programs from the Olden and MiBench benchmark suites improves on average from 2.4% to 12% depending on the latency of the functional units, and up to 39.6%; more precisely, the performance of optimized code sections improves on average from 3.5% to 19%, and up to 49%.*

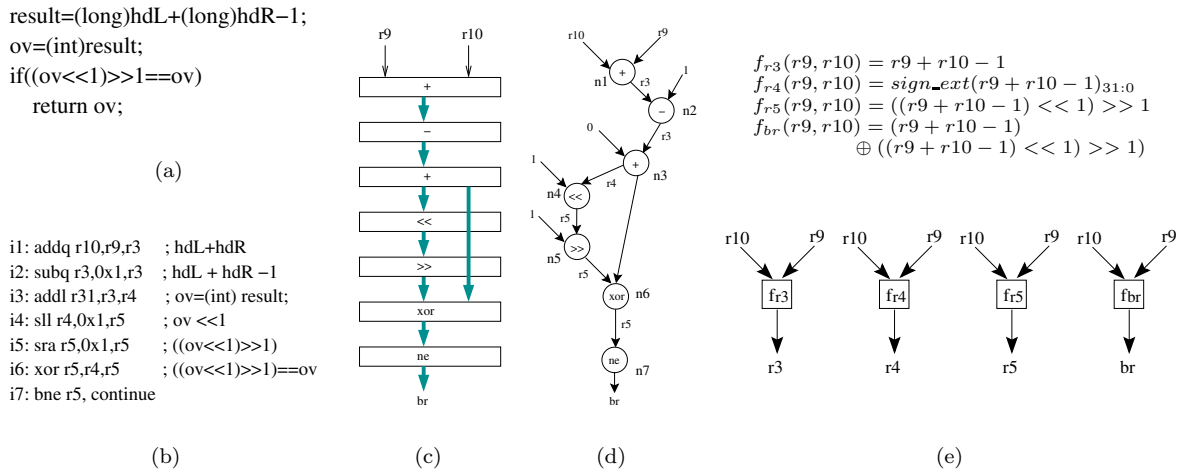
## 1. Introduction

Current and upcoming processors heavily rely on increasing instruction throughput through pipelining and exploiting all forms of ILP. The additional on-chip space which comes with each new processor version is increasingly devoted to these techniques (larger pipelines, larger branch prediction tables, larger caches, larger instruction windows and reservation stations...) rather than to computing resources themselves (functional units). Besides, throughput and ILP techniques increasingly rely on speculative mechanisms (branch prediction, instruction and data prefetching, value prediction...), and the quality of each individual prediction mechanism tends to improve slowly.

Since each mechanism comes at a significant on-chip space cost, it is not obvious that speculation will always

remain the most complexity-effective path to performance improvements. Already, two recent approaches, the Chimaera architecture [23] and the Grid Processor Architecture (GPA) [12] used in the TRIPS architecture [19] propose to use on-chip space differently to improve performance. Both approaches rely on a common principle: directly map part of the program dataflow graph to the architecture, so that instructions become hardware operators and execute much faster; Chimaera maps instructions to reconfigurable circuits, and GPA to grids of ALUs. However, once translated into hardware, a sequence of dependent instructions remains a sequence of dependent (connected) hardware operators. Therefore, both approaches are again limited by intrinsic ILP [22], and even more by the compiler ability to extract ILP [24], just like current and upcoming processors. And the increasing processor architecture complexity combined with the limitations of static analysis on pointer-based codes, like the SPECInt2000 and the Olden [18] benchmarks, already considerably strain the compiler.

In this article, we propose an approach for exploiting additional on-chip space that is not limited by the lack of ILP and that does not require complex software support. The starting point of our approach is to note that many stateless sequences of instructions can be viewed and expressed as a *function*: it has input data (the function parameters) and has output data (the value of the function). Unlike *algorithms*, *functions* can be mapped very easily to a combinational 2-level sum of products circuit (ORs of ANDs). While this transformation is extreme and its cost is prohibitive, it does show that it is possible to obtain a sequence of dependent instructions as a combinational logic circuit. Implicitly, this transformation trades on-chip space for computing resources and achieves high speed by exploiting *circuit-level parallelism*. We present an approach that exploits this principle while avoiding the exponential cost of the 2-level circuit transformation. The mechanism is implemented in a superscalar processor using a large and



**Figure 1. An example of instruction collapsing: (a) C code, (b) assembly code, (c) non-collapsed hardware operators, (d) corresponding DFG, and (e) corresponding functions.**

scalable functional unit, called the *Function* unit. Even though this unit is reconfigurable, its structure is very different and more simple than traditional FPGAs, especially with respect to its interconnection network. The functions are built offline using the trace builder presented in the rePLay framework [7], and we have implemented the corresponding toolset that automatically converts rePLay frames into mappable functions.

With this mechanism, transformed frames execute up to 49% faster for some codes. This mechanism illustrates a first implementation of an approach that provides a different way to improve the performance of sequences of dependent instructions.

In Section 2, we present the principles of the approach, the methodology in Section 3, we analyze the potential speedup and limitations of the approach in Section 4, we present the implementation and experimental results in Section 5.

## 2. Principles

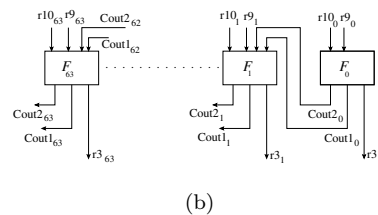
To illustrate our approach and compare it with existing solutions, we will use the example of Figure 1(a) extracted from procedure `Sum` of the SPECInt2000 benchmark 254. `gap`. This code adds two integers and compares the two most significant bits to check for overflow. The resulting assembly code on an Alpha EV6 processor is a fully sequential set of instructions, i.e., no two instructions can execute in parallel, as shown in Figure 1(b). Therefore, current and upcoming processors, which heavily rely on the exploitation of all

forms of ILP, can do little to improve the performance of such codes. The Chimaera [23] or GPA [12] approaches would map the corresponding dataflow graph of Figure 1(b) respectively to a reconfigurable circuit or a grid of ALUs. The mapped hardware operators would perform faster than a set of instructions, but they would still operate *sequentially*, as shown in Figure 1(c).

In our approach, we split the dataflow graph (DFG) of Figure 1(d) into a set of independent single-output functions, one for each output of the dataflow graph, as shown in Figure 1(e). At the cost of redundant operations, e.g.,  $r9+r10-1$ , and thus hardware resources, all these functions can execute in parallel. Now, each function can be translated into a combinational logic function and *collapsed* into a 2-level logic circuit or a LUT. However, a simple function with two parameters like `fr3`, corresponds to a  $2^{128}$ -bit truth table for each output bit (assuming two 64-bit registers), which is not realistic. One way to alleviate this size problem is to implement a function of  $n$  input bits as a set of  $n$  1-bit operators associated with a multiple-carry propagation network, as shown in Figure 2(b). Each operator can be implemented as a reconfigurable logic block much like in FPGAs, but the number of 1-bit inputs is higher than in traditional FPGAs, e.g., 4-input lookup-tables (LUTs) in the Virtex-II Xilinx architecture [2], versus 6 inputs in our implementation. On the other hand, the placement and routing are much more simple. Further increasing the number of inputs would slightly increase performance (we have evaluated the potential of

$$\begin{aligned}
(r9 + r10)_i &= \begin{cases} r9_0 \oplus r10_0 & i = 0 \\ r9_i \oplus r10_i \oplus Cout1_{i-1} & 0 < i \leq 63 \end{cases} \quad (1) \\
Cout1_i &= \begin{cases} r9_0 \cdot r10_0 & i = 0 \\ r9_i \cdot r10_i + r9_i \cdot Cout1_{i-1} \\ \quad + r10_i \cdot Cout1_{i-1} & 0 < i < 63 \end{cases} \quad (2) \\
(r9 + r10 - 1)_i &= \begin{cases} (r9 + r10)_0 \oplus 1 & i = 0 \\ (r9 + r10)_i \oplus 1 \oplus Cout2_{i-1} & 0 < i \leq 63 \end{cases} \\
&= \begin{cases} r9_0 \oplus r10_0 & i = 0 \\ r9_i \oplus r10_i \oplus Cout1_{i-1} \oplus Cout2_{i-1} & 0 < i \leq 63 \end{cases} \quad (3) \\
Cout2_i &= \begin{cases} (r9 + r10)_0 & i = 0 \\ (r9 + r10)_i + Cout2_{i-1} & 0 < i \leq 63 \\ r9_0 \oplus r10_0 & i = 0 \\ (r9_i \oplus r10_i \oplus Cout1_{i-1}) + Cout2_{i-1} & 0 < i \leq 63 \end{cases} \quad (4)
\end{aligned}$$

(a)



(b)

**Figure 2. Translating function  $r3$  into a hardware operator: (a)  $r3$  function, and (b) 64 1-bit operators with multiple-carry propagation.**

up to 40-input blocks), but it would also significantly increase operators size, see Section 4. The 1-bit logical expressions associated with function  $fr3$  are shown in Figure 2(a). Assuming a collapsed sequence of instructions in a *Function* unit executes as fast as a single instruction, the length of the sequence of dependent instructions is an upper-bound of the speedup, i.e., 7 in the example of Figure 1. The impact of the *Function* unit delay on performance is studied in Section 5.4.

The notion of collapsing instructions was previously introduced by Phillips et al. who proposed a 3-1 interlock collapsing adder that could collapse two dependent adds into one specific 3-input adder [16]. They later investigated the potential of collapsing up to three dependent instructions [20], but neither the concept nor its implementation were generalized, and the notion of functions was not introduced in these studies. Similarly, an instruction **Scale and Add**, which adds an operand to another multiplied by a factor, is implemented in the Alpha ISA [1]. Furthermore, to a certain extent, Chi-maera [23] proposes a limited form of instruction collapsing by combining arithmetic operations, e.g., ADD, with bit-shifting instructions: in fact, a single arithmetic operation takes place on each row of the reconfigurable unit but the interconnection network between rows is used to implement the shifts, hence the collapsing. The notion of “function” is implicitly widely used in ASIC and ASIP [9], but not in a way that can be applied to general-purpose processors. PRISC [17] proposes to map operations on hardware-programmable functional units (PFUs), and recently, Clark et al. [5] proposed a method to automatically extract candidate functions from programs, but, in both approaches, the operations are not extracted at run time, and cannot

span across multiple basic blocks.

Our approach has three major assets: (1) in theory it applies to almost any sequence of dependent instructions, (2) it doesn’t rely on ILP exploitation, and (3) translating DFGs into functions requires only straightforward transformations.

### 3. Experimental Framework

We performed two sets of experiments: architecture-independent experiments which aim at determining the potential of the approach, see Section 4, and experiments on a superscalar processor coupled with the rePLay framework without optimization (only the frame builder is used), see Section 5. We developed a specific toolset to translate Alpha assembly instructions into circuit configuration macros. It can be implemented either as a static compilation tool, a dynamic compilation tool, or in hardware. On purpose, we developed a fully automatic toolset in order to demonstrate that the added compiler and hardware complexity can be harnessed.

**Translating dependent instructions into configuration macros.** The four phases of our optimization engine are shown in Figure 3: first, we dynamically split the program execution trace into large chunks of consecutive instructions that we call a *trace*, and we apply the next steps to each trace. Note that the trace size is fixed for the architecture-independent experiments, and is variable in the superscalar processor experiments, see Section 5. Next, data dependencies in the trace are analyzed and the dataflow graph (DFG) for that trace is built as in Figure 1(d). Then functions are selected within the DFG modulo the rules described in Section 4.2. Finally, the truth table associ-

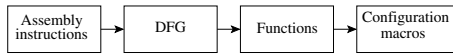


Figure 3. Phases of the optimization engine.

ated with each bit of the function is computed as well as the associated carry chain functions described in Section 2 and Figure 2. The LUT (Look-Up Table) configurations directly derive from the truth tables. The algorithm and its implementation are detailed in Section 5.

**Simulation methodology.** In all experiments we used the SimpleScalar emulator [3] of the Alpha ISA, the SPECInt2000 benchmarks, as well as 9 of the Olden benchmark suite [18] (bh, em3d, health, mst, perimeter, power, treadd, tsp and bisort) and 3 of the MiBench suite [10] (patricia, tiff2bw and djpeg). We tested 100 million consecutive instruction traces for each benchmark, focused on the most time-consuming procedures (selected using profiling on full benchmarks executions). The benchmarks were compiled using the Compaq Alpha compiler with full optimizations (`-fast`). For the superscalar processor experiment, we used the `sim-outorder` architecture [21] and applied our transformations to rePLay frames.

## 4. Potential of the Approach

### 4.1. Potential performance improvements

To evaluate the potential of the approach, we want to compute the theoretical speedup over an idealized processor where all instructions that *can* execute in parallel *do* execute in parallel. Thus performance improvements only come from executing sequential instructions as collapsed functions. The idealized processor is defined as having a 1-cycle ideal memory, perfect branch prediction, infinite instruction window, issue width, and reservation stations.

As explained in Section 2, the potential speedup is, in theory, determined by the number of dependent instructions collapsed, i.e., 7 in the example of Figure 1. However, consider the DFGs of Figure 4. DFG1 in Figure 4(a) represents a sequence of 7 dependent instructions, like our example of Figure 1, resulting in a theoretical speedup of 7. DFG2 in Figure 4(b) contains again 7 instructions but it has 3 branches, one per out-function, and the largest branch of the DFG contains 4 instructions, thus, the maximum speedup is 4 in this case. Similarly, for DFG3 in Figure 4(c), the theoretical speedup is 2, again with 7 instructions. Therefore, to compute the theoretical speedup of a trace of

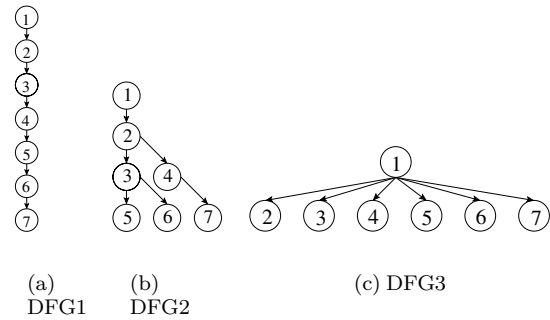


Figure 4. Different possible DFG shapes.

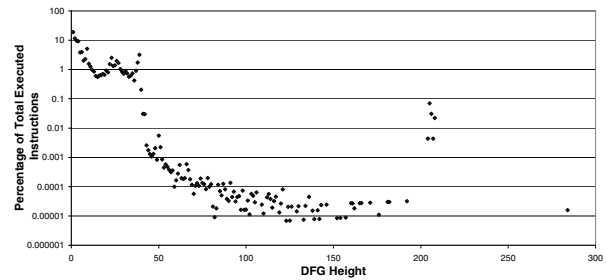


Figure 6. DFG height distribution.

instructions, we need to identify all *disjoint* DFGs in the trace, i.e.,  $DFG_a$  is disjoint from  $DFG_b$  if none of the instructions of  $DFG_a$  depends on an instruction in  $DFG_b$  and reciprocally. Then, the theoretical speedup of a DFG is equal to the size of its largest branch, or, in other terms, to its critical path or height. Thus, the traces are partitioned into disjoint DFGs, and the theoretical speedup of each DFG is calculated. The theoretical speedup of a program trace is the average height of all DFGs in the trace. An important limiting factor for the speedup is the trace size. The larger the traces, the larger the DFGs and the speedup. Using 1024-instruction traces and up to 40-input operators, the average theoretical speedup for all benchmarks is 1.5 with a maximum speedup of 2.32 for the `djpeg` benchmark, see Figure 5. Figure 6 shows the distribution of the height of DFGs as a percentage of the total number of instructions executed, also using traces of 1024 instructions, averaged on all benchmarks. While there are very large DFGs, i.e., over 250 instructions, many DFGs are rather small. The factors limiting the size of the DFGs and how these limitations can be overcome are discussed in Section 4.2.

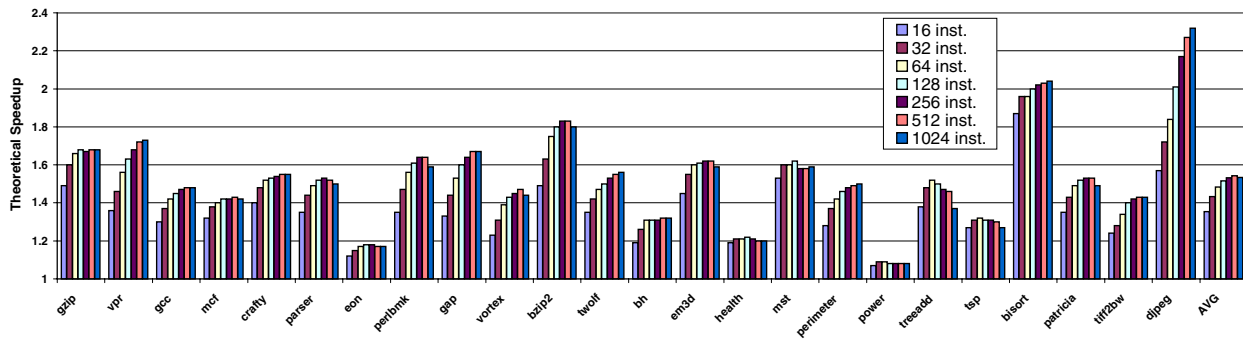


Figure 5. Theoretical speedup for different trace sizes.

#### 4.2. Analyzing and overcoming the limitations of the approach

In theory, the whole program can be one huge DFG. In practice, DFGs need to be split due to many factors, which we call DFG *cuts*. A *cut* is an instruction that prevents further collapsing and thus reduces speedup opportunities. It is then transformed into a function output of its parents' DFG and becomes an input for its children's DFGs. Besides these "true" cuts, the trace size limitation mentioned in Section 4.1 introduces additional methodology-related cuts. The different types of "true" cuts are discussed below.

**Number of function inputs.** We call *physical inputs* both the register inputs and the inputs corresponding to carries, see Figure 2(b). The maximum number of physical inputs per function determines the size of the 1-bit hardware operators used to implement functions. Since the hardware operator size is fixed, the maximum number of inputs is fixed as well, and any DFG requiring more than the maximum number of inputs must be cut. Figure 7 shows the cumulative distribution of the number of inputs per function, averaged over all benchmarks, using 1024-instruction traces. We observed that more than 80% of the functions require fewer than 10 physical inputs, so that implementing even large functions does not require large 1-bit operators. In our implementation we have used 6-input logic blocks. Figure 8 confirms that increasing the number of inputs beyond 10 has a negligible impact on the theoretical speedup.

**Load instructions.** For the moment, *loads* induce cuts because they cannot be combined with subsequent cuts because they cannot be combined with subsequent dependent instructions, though we are currently investigating several ways to alleviate these cuts such as data preloading. While, on average, 24.43% of executed instructions are load instructions, their irregular occurrence in DFGs still enables large DFGs, as shown in

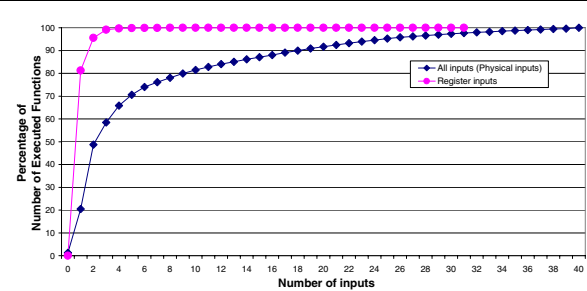


Figure 7. Cumulative distribution of the number of inputs per function.

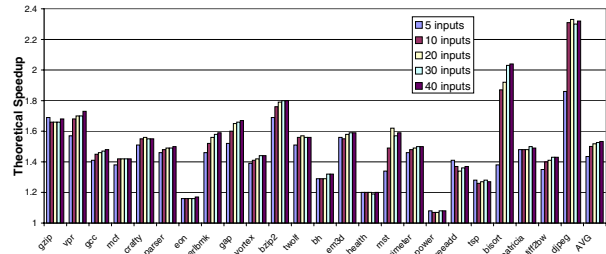


Figure 8. Impact of the number of inputs on the theoretical speedup.

Figure 6. Store and branch instructions are not *cuts*, they are exit points of DFGs. Load and store instructions are still considered collapsible: address computation instructions can be collapsed with loads and stores, and value computation instructions can be collapsed with stores. Our transformation engine detects pairs of statically dependent store/load and replaces such "load



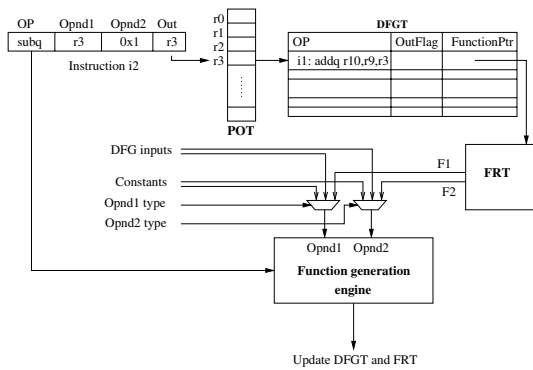


Figure 12. Function generation.

in the process of building functions, and we illustrate this process with the example of Figure 1.

**Building the DFG.** Each instruction in the trace, i.e., the instructions of Figure 1(b), are loaded in the DFGT (*DataFlow Graph Table*), with one entry per instruction, see Figure 12.<sup>1</sup> Consider instruction `i1: addq r10,r9,r3`: when the instruction is stored in the DFGT, the flag `OutFlag` is set to indicate the instruction is an output of the DFG being progressively built; when instruction `i2: subq r3,0x1,r3` is loaded, this flag will be reset because instruction `i1: addq r10,r9,r3` will no longer be a DFG output.

In the same time, we keep track of the data dependencies between instructions through the *Producing Output Table* (POT). The POT is indexed by the instruction destination register. Each entry contains an index to the DFGT, i.e., to the instruction that produces the corresponding register. For instruction `i1`, since `r9` and `r10` are inputs to the DFG, i.e., they are not generated by another DFG instruction, the corresponding entries in the POT will not point to a DFGT entry. The combined role of the POT and the DFGT is akin to the role of the ROB in a superscalar architecture, except that we cannot use the processor ROB to perform that task since function building is performed *offline* by the rePLAY framework.

**Generating the function corresponding to an instruction.** Once a new DFG instruction is loaded in the DFGT, we build the function corresponding to the instruction operation, or more exactly, we compose this operation with the functions producing its source operands, creating a more complex function. For that purpose, we send the instruction source operands to

1 Note that complex instructions, such as the Alpha Scaled Add instruction `s4addq`, are decomposed into several elementary operations, each corresponding to a DFG node, and thus to an entry in the DFGT.

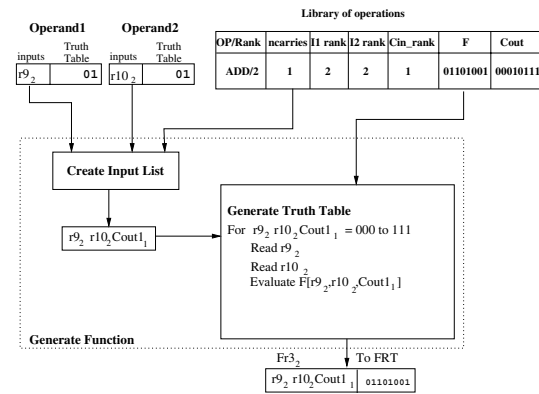


Figure 13. Function Generation Engine: generating bit 2 of node `n2`.

	F	Cout1	Cout2	...	Coutn
$F_{i2}$					
bit0	$r9_0, r10_0$	0110	$r9_0, r10_0$	0001	
bit1	$r9_1, r10_1, Cout1_1$	01101001	$r9_1, r10_1, Cout1_1$	00010111	
...					
bit59	$r9_{59}, r10_{59}, Cout1_{59}$	01101001	$r9_{59}, r10_{59}, Cout1_{59}$	00010111	

Figure 14. Function Repository Table (FRT).

the *Function Generation Engine* (FGE), see Figure 13. There are three types of operands, DFG inputs which are operands not produced by other instructions in the DFG, constant inputs, and operands produced by other instructions in the DFG. The first two types are simply sent to the Function Generation Engine. If the operand is produced by another instruction, then we send the function producing this operand as a truth table. Functions are stored in the FRT (*Function Repository Table*) as 64-bit truth tables (for a maximum of 6 inputs for each function), one per word bit, see Figure 14. Each DFGT entry (instruction) contains an index to the corresponding function in the FRT. Besides the truth table, the FRT also contains the number of inputs of the truth table. Each bit of the operand is a (*input list, truth table*) pair. When the operand is just a value, as for registers `r9` and `r10` of instruction `i1`, the number of inputs is 1 and the truth table is the identity function (01).

To create the truth table of the composed function (the operation of the current instruction composed with the functions creating the operands), the Function Generation Engine performs as follows. For each possible combination of all input variables (the input variables of both operands), the engine looks up the truth tables of the operands, and uses these values to

then look up the truth table of the operation itself. The truth table of the operation is stored in a library of operations in the Function Generation Engine. Besides the truth table, the library also indicates if additional function variables must be introduced, such as (and usually) carries. For instance, for instruction `i1`, the operands are `r9` and `r10` with identity truth tables. After looking up the library of operations, the engine sees that the carry must be one of the function inputs since the operation is an addition, so there are three input variables:  $r9_k r10_k C_{out1_{k-1}}$  for bit  $k$  of the composed function. Then, the truth table of the composed function (in this case, just an addition) is output (in this case, the adder truth table). Function truth tables are then stored in the FRT, and the corresponding index is stored in the DFGT. The whole process for instruction `i2:subq r3,0x1,r3` is similar except that the POT entry corresponding to `r3` will point to the `i1` entry in the DFGT. The function corresponding to this entry will thus be passed to the Function Generation Engine as well as the constant `0x1`.

**Cuts.** Input operands that are dependent on instructions that were identified as *cuts* (see Section 4) are treated as identity functions also.

**Filtering functions.** Depending on the latency of the *Function* unit, a sequence of collapsed instructions may execute slower in the *Function* unit than in the normal execution units, if the number of collapsed instructions is small. For example, if a function is collapsing three dependent 1-cycle latency instructions, the *Function* unit should execute in less than three cycle to execute faster. Assuming a 1-cycle ALU latency, we *filter* candidate functions by selecting only those disjoint DFGs (see Section 4) with height greater than the *Function* unit latency. Instructions that do not belong to these selected DFGs are marked as non-collapsible, and are executed in the normal execution units. This heuristic allows better utilization of all functional units and prevents non-appropriate functions from slowing down the execution.

## 5.2. Hardware implementation of functions

Figure 15 shows the implementation of functions as an additional large functional unit. As explained in Section 2, we implement functions using a set of 64 1-bit chained operators. These operators represent one of the bits of an  $n$ -input function, as explained in Section 2. Since the functions vary constantly from one trace to another, we use reconfigurable logic to implement the 1-bit operators. However, it is important to note that our operators need not bear the same limitations as traditional FPGAs: (1) the chained oper-

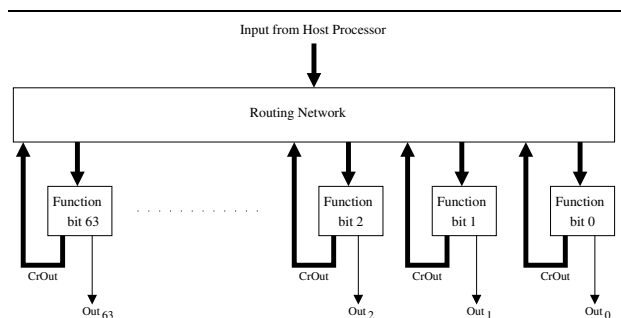


Figure 15. Implementation of functions.

ators only contain combinational logic, no sequential logic is necessary, and (2) only one row of operators is needed. The operators are linked in an unusual but simple manner: *multiple* carries are propagated from operator  $i$  to higher order bits only, therefore avoiding the complex interconnection networks that usually account for more than 90% of on-chip space in FPGA circuits [6]. On the other hand, our approach relies on functions with a significant number of inputs, resulting in larger logic blocks. As mentioned in Section 4.2, we assumed a maximum of 6 physical inputs for each bit of the *Function* units.

## 5.3. Implementing functions using the rePLAY hardware framework

The two major implementation issues of our approach are the overhead of dynamically building DFGs and functions on-the-fly, during execution, and assembling large traces. The rePLAY environment proposed by Patel et al. [7][14][15] can partially address both issues.

The rePLAY framework provides a dynamic optimization support for building large traces of instructions (frames) after retirement. Moreover, the frames are transformed offline, i.e., out of the critical path. We implemented the rePLAY architecture framework, augmented with our function optimization engine and the associated *Function* units, in the SimpleScalar simulation environment. We assumed a future scaled-up 8-way superscalar processor architecture, see Table 1 for the modified parameters. Figure 16 shows the core processor together with rePLAY and the function mechanism which includes several *Function* units. Our optimization mechanism can be built on top of the optimizations proposed in [7] which improve ILP while our techniques focus on ILP-deprived code sections. To outline the impact of the *Functions* mechanism, we implemented rePLAY without frame optimizations; thus

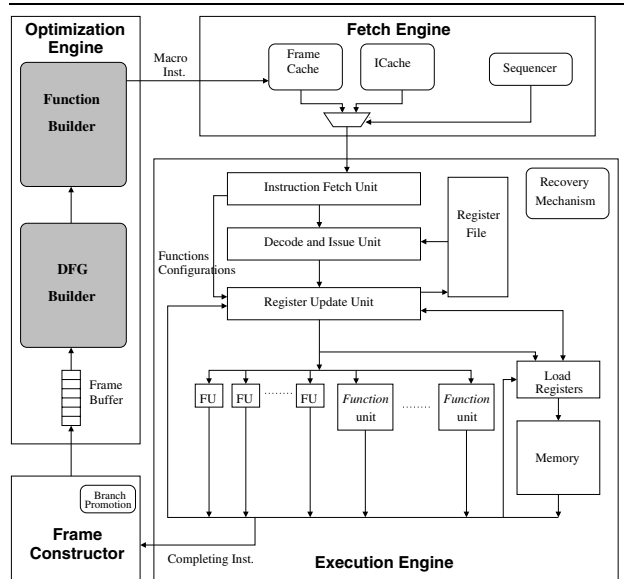
Fetch width	16
Issue / Decode / Commit width	8
RUU size (Inst. window- ROB)	1024
LSQ size	128
ExeUnits	8 IALU, 4 IMULT, 4 FPALU, 4 FPMULT
<i>Function</i> units	8
Branch	Combined, 4K entries bimodal, and 2 level Gap predictor, 8K 2nd level entries, 14 history wide, 1K meta-table size 7 cycle BR resolution
Memory Latency	70 cycles
L1 DCache	32kB, 1 cycle
L1 ICache	16kB, 1 cycle
L2 Unified Cache	1MB, 6 cycle

**Table 1. Baseline configuration of the processor core.**

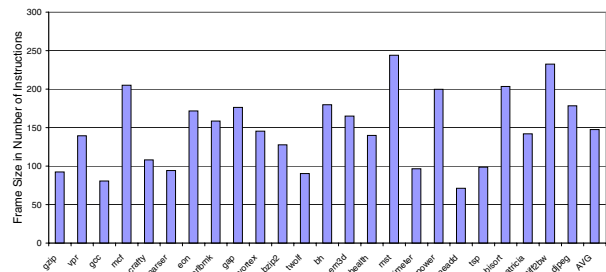
the performance improvements reported in Section 5.4 solely correspond to the *Functions* mechanism.

The rePLay framework collects traces of committed instructions to form “frames”. The frames are frequently executed sequences of instructions. The *frame constructor* adds each committed instruction into a frame construction buffer, until a branch with a non-highly stable behavior (taken/not taken for conditional branches or constant target addresses for indirect branches) is encountered. Branches with highly stable behavior are called promoted branches in the rePLay framework [13], and are stored in a *branch bias table*. When a non-promoted branch is encountered or when the frame reaches a maximum size of 256 instructions, the frame is passed to the optimization engine to build functions (frames smaller than 32 instructions are discarded to avoid saturating the optimization engine). Our back-end transformation engine forms the DFG for each frame and transforms the trace of instructions into a trace of functions or *macro-instructions*, see Section 5.1.

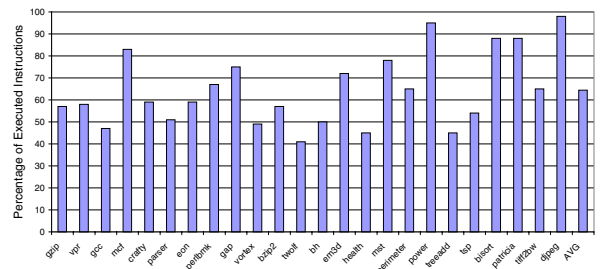
Because it is difficult to estimate a priori the exact delay of the optimization engine (whether implemented in hardware or software), we assumed a 1000-cycle optimization engine delay. Fahs et al. showed in [7] that the optimization delay may have very little impact on performance. We tested a 10000-cycle delay and only ob-



**Figure 16. The core architecture.**



**Figure 17. Average frame size.**



**Figure 18. Dynamic instructions coverage.**

served an average performance slowdown of less than 1%. The generated functions are cached into the frame cache and are directly forwarded to the *Function* units upon a frame cache hit. *Function* units may be config-

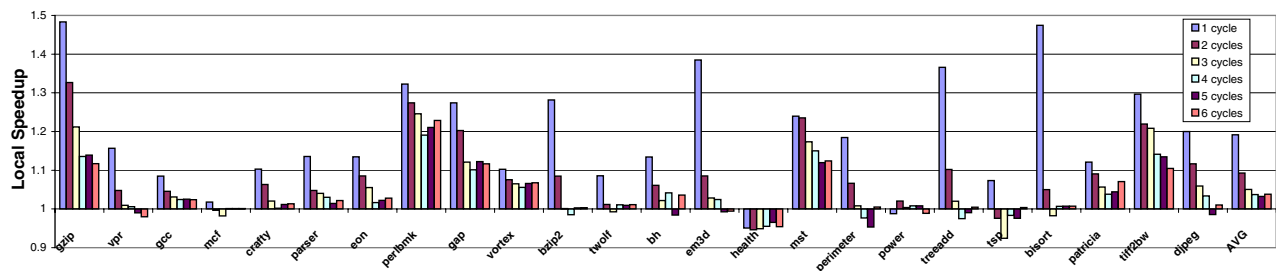


Figure 19. Local Speedup.

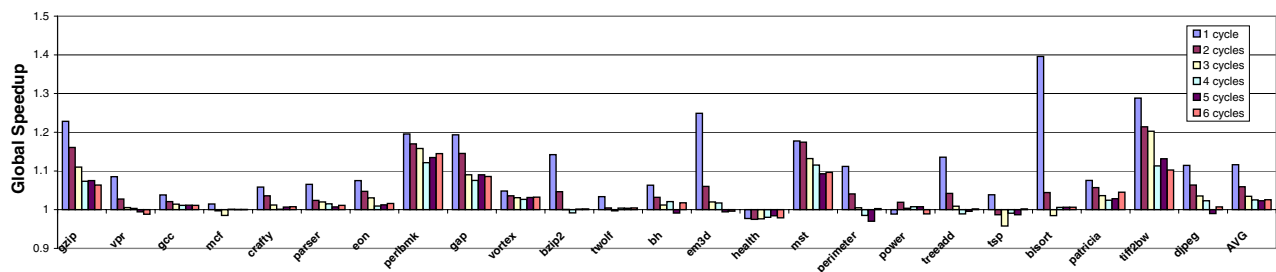


Figure 20. Global Speedup.

ured while they are scheduled provided there is a sufficient number of functional units, as originally proposed in the PipeRench architecture [4]. An important aspect of the rePLAY framework is that, once dispatched, the frames should be run to completion. Therefore, branch instructions in frames are replaced with *assertions*. When an assertion is not verified, rePLAY provides a mechanism to revert the architectural state to the beginning of the executed frame. We modeled this mechanism using a 10-cycle penalty. The replacement of branches by assertions fits well our approach. Since each conditional branch is transformed into a 1-bit function, as shown in the example of Figure 1, collapsing the function can speed up the branch resolution which, in turn, can reduce the frame misprediction penalty. More generally, collapsing functions corresponding to branch conditions can speed up branch resolution, using similar principles but a different technique than *Anticipation* [8].

We parameterized the rePLAY environment as follows: a 32K-entry bias table for direct branches, a 4K-entry bias table for indirect branches, a 14-branch history is used as a hashing key to the bias table, a branch is promoted to a highly biased branch after a threshold of 16 consecutive stable behavior, a 128-frame buffer to store frames while they are processed, a 4-way associative frame cache [7] that can store up to 4K frames and

128K macro-instructions, each frame in the frame cache is tagged using a history path of 4 branches. Upon hit, the frame is dispatched, otherwise standard instructions are fetched from the instruction cache and dispatched.

#### 5.4. Performance Analysis of the Implementation

*Efficiency of frame building.* The main asset of rePLAY is its ability to dynamically build large sequences of instructions. We experimentally observed that the average frame size is 147 instructions, see Figure 17. On average, 65% of all instructions instances effectively belong to optimized frames, see Figure 18. Instruction coverage is limited by non-highly biased branches, too small frames (less than 32 instructions), frame cache misses and mis-speculated frames.

*Speedup achieved with the function mechanism.* Consequently, we distinguish the speedup achieved on the transformed traces, i.e., the *local speedup* and the *global speedup*. We use the scaled-up 8-way superscalar architecture (see Table 1) coupled with rePLAY as the baseline configuration. We have yet to implement a hardware model of the *function* unit at the circuit level to precisely estimate its latency. Even then, there are multiple possible architecture choices: since these

chained operators implement multiple but straightforward carry propagation, they can benefit from the fast but complex carry-propagation schemes that were specifically designed for speeding up FPGA-based carry chains [11], and which are different from the standard high-performance adder carry chains [25]. Therefore, we varied the latency of the *Function* unit from 1 to 6 cycles. Assuming a 1-cycle *Function* unit, Frames transformed into functions execute 19% faster than the baseline configuration on average, with a maximum of 49% for the `gzip` benchmark, as shown in Figure 19. Because of the still limited coverage of the rePLay environment, the global performance improvement is only 12% but with strong variations up to 39.6% for the `bisort` benchmark, see Figure 20. Codes with large sets of sequential and dependent instructions particularly benefit from the mechanism. The low speedups are mainly due to high percentage of non-collapsible instructions (`eon` and `power`) or long chains of dependent loads that limit the height of collapsed instructions (`mcf` and `health`).

## 6. Conclusions and Future Work

In this paper, we presented an approach for improving the performance of sequences of dependent instructions by expressing these sequences as functions, collapsing them into hardware operators and taking advantage of circuit-level redundancy. Our approach does not rely on ILP exploitation, and the associated software optimizations are fairly simple.

We tested an implementation of this approach on a scaled-up superscalar processor with the rePLay framework, and we observed an average performance improvement varying from 3.5% to 19% on optimized code sections.

We are currently investigating the coupling of the function mechanism with address prediction mechanisms to remove many of the *cuts* due to load instructions. Removing some of the cuts will increase the average function size and the overall speedup.

## Acknowledgements

We would like to thank the anonymous reviewers for their helpful insights and suggestions. Also we would like to thank all the members of the ALCHEMY research group at INRIA-Futurs, especially Alexandre Farcy and Daniel Gracia Pérez for their useful feedbacks on earlier versions of this article.

## References

- [1] Alpha architecture handbook, October 1998. Compaq Computer Corporation.
- [2] Virtex-II pro platform FPGAs: Functional description. Technical Report DS083-2, January 2002. Xilinx Corporation.
- [3] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The SimpleScalar tool set. Technical Report CS-TR-1996-1308, 1996.
- [4] Y. Chou, P. Pillai, H. Schmit, and J. P. Shen. PipeRench implementation of the instruction path coprocessor. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 147–158. ACM Press, 2000.
- [5] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *Proceedings of the 36th annual ACM/IEEE international symposium on Microarchitecture*, 2003.
- [6] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (CSUR)*, 34(2):171–210, 2002.
- [7] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. J. Patel, and S. S. Lumetta. Performance characterization of a hardware framework for dynamic optimization. In *Proceedings of the 34th annual IEEE/ACM international symposium on Microarchitecture*. ACM Press, December 2001.
- [8] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 59–68. IEEE Computer Society Press, 1998.
- [9] D. Fischer, J. Teich, M. Thies, and R. Weper. Efficient architecture/compiler co-exploration for ASIPs. In *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems*, pages 27–34. ACM Press, 2002.
- [10] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, December 2001.
- [11] S. Hauck, M. Hosler, and T. Fry. High performance carry chains for FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(2), April 2000.
- [12] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, 2001.
- [13] S. J. Patel, M. Evers, and Y. N. Patt. Improving trace cache effectiveness with branch promotion and trace packing. In *Proceedings of the 25th annual international symposium on Computer architecture*, pages 262–271. IEEE Press, 1998.

- [14] S. J. Patel and S. S. Lumetta. rePLay: A hardware framework for dynamic optimization. *IEEE Transactions on Computers*, 50(6), June 2001.
- [15] S. J. Patel, T. Tung, S. Bose, and M. M. Crum. Increasing the size of atomic instruction blocks using control flow assertions. In *Proceedings of the 33rd annual IEEE/ACM international symposium on Microarchitecture*, pages 303–313. ACM Press, 2000.
- [16] J. Phillips and S. Vassiliadis. High-performance 3-1 interlock collapsing ALU's. *IEEE Transactions on Computers*, 43(3), March 1994.
- [17] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 172–180. ACM Press, 1994.
- [18] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(2):233–263, 1995.
- [19] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 422–433. ACM Press, 2003.
- [20] Y. Sazeides, S. Vassiliadis, and J. E. Smith. The performance potential of data dependence speculation & collapsing. In *Proceedings of the 29th annual IEEE/ACM international symposium on Microarchitecture*, pages 238–247. IEEE Computer Society Press, 1996.
- [21] G. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39(3), March 1990.
- [22] D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 176–188. ACM Press, 1991.
- [23] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 225–235, Vancouver, British Columbia, June 12–14, 2000. IEEE Computer Society and ACM SIGARCH.
- [24] Z. A. Ye, N. Shenoy, and P. Banerjee. A C compiler for a processor with a reconfigurable functional unit. In *Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, pages 95–100. ACM Press, 2000.
- [25] M. Ziegler and M. Stan. Optimal logarithmic adder structures with a fanout of two for minimizing the area-delay product. In *Proceedings of the the International Symposium on Circuits and Systems*, May 2001.