From Sequences of Dependent Instructions to *Functions*: A Complexity-Effective Approach for Improving Performance without ILP or Speculation

Sami YEHIA and Olivier TEMAM
LRI, Paris South University, France
{yehia,temam}@lri.fr

Abstract

In this article, we present a new approach for improving the performance of sequences of *dependent* instructions. We observe that any sequence of instructions can be interpreted as a *function*. Unlike sequences of instructions, functions can be translated into very fast but exponentially costly two-level combinatorial circuits. We present an approach that exploits this principle, speeds up programs thanks to circuit-level parallelism/redundancy, but avoids the exponential costs. Current superscalar processor architectures scale by further exploiting ILP and adding the necessary harware elements. As the superscalar degree increases, fewer programs or program sections will benefit from this approach, and it will become more complexity-effective to devote hardware space to ILP-less program sections.

We analyze the potentials of this approach, and then we propose an implementation that consists in a superscalar processor with a large specific functional unit associated with specific backend transformations. On average, the performance of SpecInt2000 benchmarks improves by 11.5% and up to 28%; more precisely, the performance of optimized code sections improves by 22% on average, and up to 58%.

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). However 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 [18] and the Grid Processor Architecture (GPA) [10] 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 [17], and even more by the compiler ability to extract ILP [19], 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 benchmarks, already considerably strain the compiler.

In this article, we propose a CE 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 any sequence of instructions and even a whole program, i.e., any *algorithm*, 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 combinatorial 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 combinatorial 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 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 [6], and we have implemented the corresponding back-end that automatically converts rePLay frames into mappable functions.

With this mechanism, transformed frames execute 22% faster on average, and up to 58% 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 in Section 5, and experimental results in Section 6.

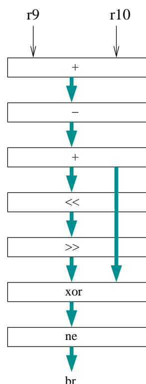


Figure 1: *Non-collapsed hardware operators*

2 Principles

To illustrate our approach and compare it with existing solutions, we will use the example of Figure 2(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 2(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 [18] or GPA [10] approaches would map the corresponding dataflow graph of Figure 2(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.

In our approach, we split the dataflow graph (DFG) of Figure 2(c) into a set of independent single-output functions, one for each output of the dataflow graph, as shown in Figure 2(d). 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 combinatorial logic function and *collapsed* into a 2-level logic circuit. 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 3(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 10 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, but it would also significantly increase operators size, see Section 4. The 1-bit logical expressions associated with function `fr3` are shown in Figure 3(a). In this first study, we did not use the faster but more complex carry-propagation schemes that were specifically designed for speeding up FPGA-based carry chains [9], and which are different from the standard high-performance adder carry chains [20].

Using the chained operators described above, the execution time of our initial example is equal to the carries propagation time through the n chained operators. Assuming the propagation delay is similar to that of an n-bit adder, the theoretical speedup would be equal to 7, i.e., the length of the sequence of dependent instructions; we further analyze the impact of the propagation delay in Section 6.

The notion of collapsing instructions was previously introduced by Vassiliadis who proposed a 3-1 interlock collapsing adder that could collapse two

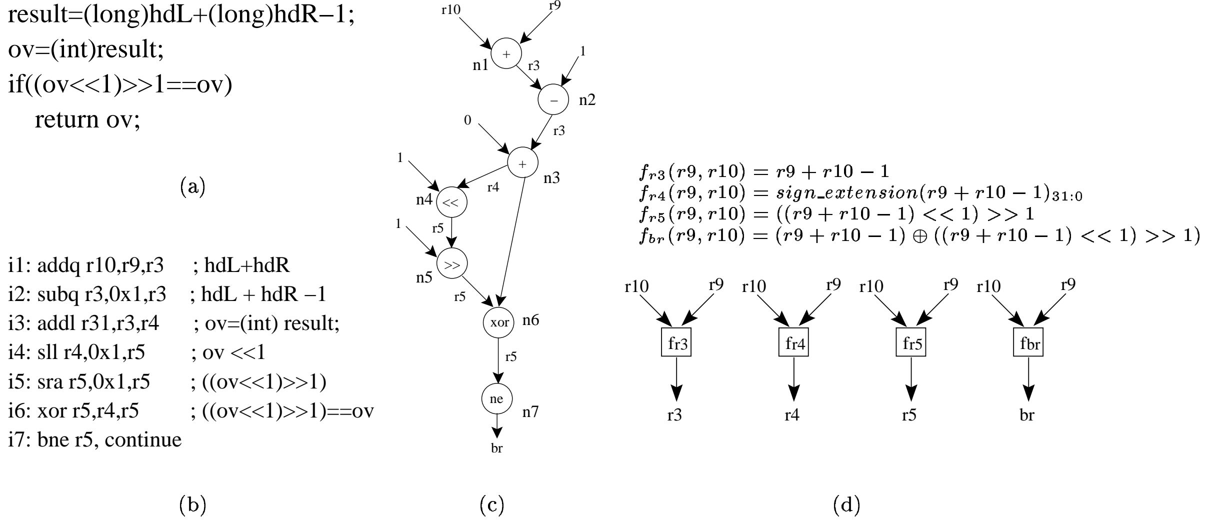


Figure 2: *An example of instruction collapsing: (a) C code, (b) assembly code, (c)corresponding DFG, (d) corresponding functions.*

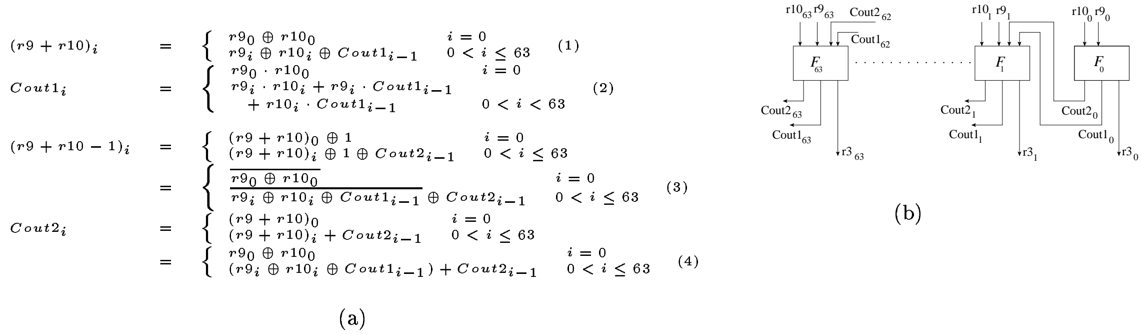


Figure 3: *Translating function* r3 *into a hardware operator: (a)* r3 *function, (b) 64 1-bit operators with multiple-carry propagation.*

dependent adds into one specific 3-input adder [14]. He later investigated the potential of collapsing up to three dependent instructions [15], 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, Chimaera [18] 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 [8], but not in a way that can be applied to general-purpose processors.

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

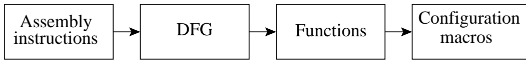


Figure 4: *Back-end chain.*

processor coupled with the rePLay framework without optimization (only the frame builder is used), see Section 5. We developed a specific compilation back-end 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 even in hardware, as we assumed in our case, see Section 5. On purpose, we developed a fully automatic toolset in order to demonstrate that the added compiler and hardware complexity can be harnessed. The four phases of our optimization engine are shown in Figure 4: 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 2(c). Then functions are selected within the DFG modulo the rules described in Section 4.2. Finally, the truth table associated with each bit of the function is computed as well as the associated carry chain functions described in Section 2 and Figure 3. The LUT (Look-Up Table) configurations directly derive from the truth tables.

The DFG nodes and the configuration macros are created incrementally throughout all the optimization chain presented in Figure 4. For example, to generate the configuration macro of `fr3` (the output of node `n2` in Figure 2(c)), the node `n1` is created from instruction `i1` and its associated function and carries are generated as in Equation (1) and (2) of Figure 3(a). Next, the node `n2` is generated from the instruction `i2`, along with its dependency on `i1`. Thus, the collapsed instructions are not necessarily consecutive but dependent. Finally, the function `fr3` is generated from the new input values (the constant 1 in this case) and the previously generated function as shown in Equation (3) and (4) of Figure 3(a). A flag associated with each node identifies whether the configuration macro is an output of the DFG or not. For example, the flag associated with `n1` is turned off when `n2` is created because of the WAW dependency between `i1` and `i2`. To facilitate function collapsing, we transform function expressions so that each output either depends on

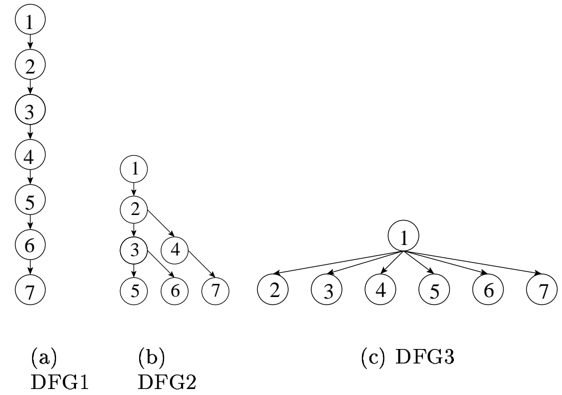


Figure 5: *Different possible DFG shapes.*

input values (a set of registers) and/or carries from the lower significant bits of DFG parent nodes. For example, the term $(r9 + r10)_i$ of Equation (3) in Figure 3(a) is directly expressed as a function of $r9_i$, $r10_i$, and the carry generated from lower significant carry $Cout1_{i-1}$ and is substituted as such in subsequent dependent functions.

In all experiments we used the Simplescalar emulator [3] of the Alpha ISA, the SPECInt2000 benchmarks, and 100 million consecutive instruction traces focused on the most time-consuming procedures selected using profiling on full benchmarks executions. The benchmarks where compiled using the Compaq Alpha compiler with full optimizations (`-fast`). For the superscalar processor experiment, we used the `sim-outorder` architecture [16] 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 2. However, consider the DFGs of Figure 5.

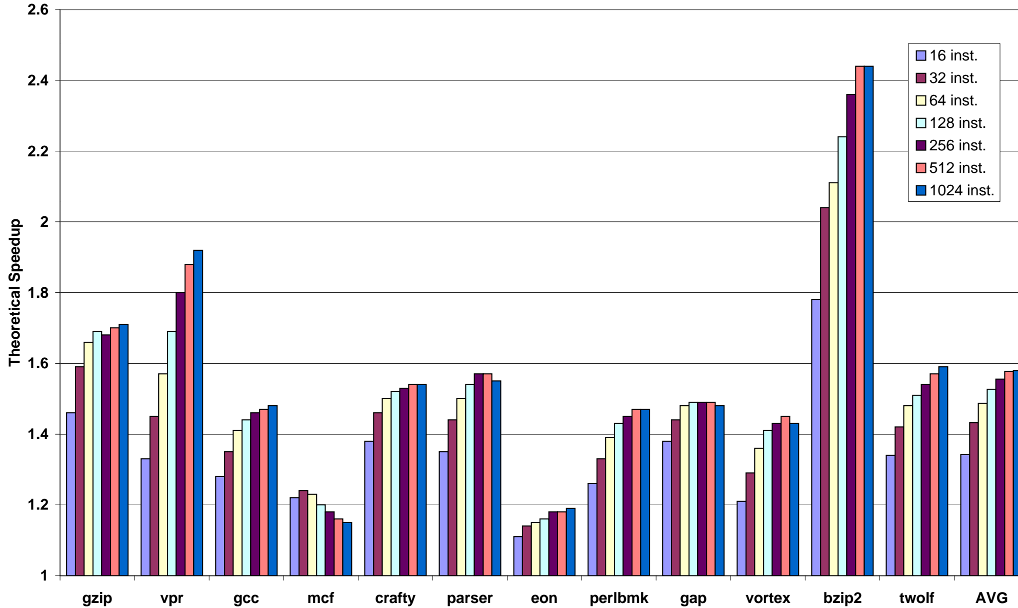


Figure 6: *Theoretical speedup for different trace sizes.*

DFG1 in Figure 5(a) represents a sequence of 7 dependent instructions, like our example of Figure 2, resulting in a theoretical speedup of 7. DFG2 in Figure 5(b) contains again 7 instructions but it has 3 branches, one per output 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 5(c), the theoretical speedup is 2, again with 7 instructions. Therefore, to compute the theoretical speedup of a trace of 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. 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 1.57 with a maximum speedup of 2.44 for the `bzip2` benchmark, see Figure 6. Figure 7 shows the distribution of the height of DFGs as a percentage of the total number of instructions executed, also using traces of 1024 instructions. While there are very large DFGs, i.e., over 250 instructions, many DFGs are rather small.

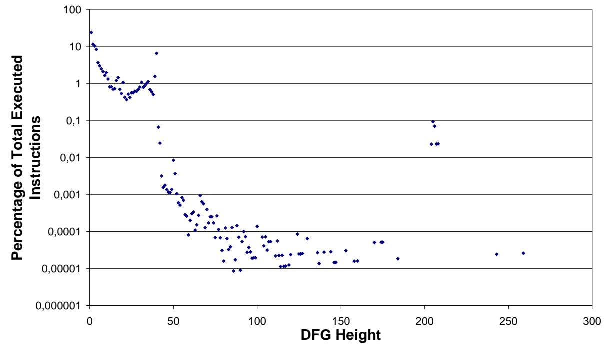


Figure 7: *DFG height distribution.*

The factors limiting the size of the DFGs and how these limitations can be overcome are discussed in Section 4.2.

4.2 Analyzing and overcoming the limitations of the approach

In theory, the whole program is 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 childs' 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 dis-

cussed below.

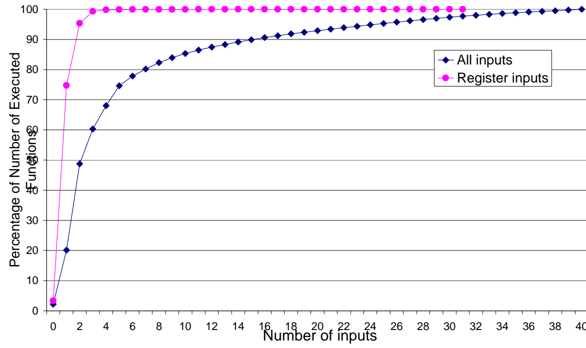


Figure 8: *Cumulative distribution of the number of inputs per function.*

Number of function inputs. We call *physical inputs* both the register inputs and the inputs corresponding to carries, see Figure 3(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 8 shows the cumulative distribution of the number of inputs per function, averaged over all benchmarks, using 1024-instruction traces. We observed that more than 85% of the functions require less than 10 physical inputs, so that implementing even large functions does not require large 1-bit operators. Figure 9 confirms that increasing the number of inputs beyond 10 has a negligible impact on the theoretical speedup.

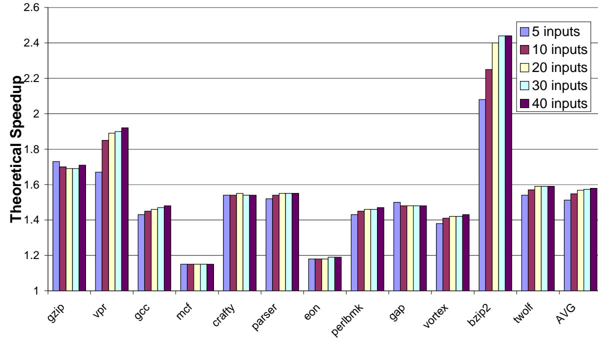


Figure 9: *Impact of the number of inputs on the theoretical speedup.*

Non-collapsable instructions. Like all other RISC codes, the Alpha binary code contains many instructions that cannot be collapsed (e.g., system calls), or which correspond to very costly hardware operators (e.g., floating-point divide). All these instructions are DFG cuts. For the moment, we only collapse integer instructions like add/sub, shift by constants, bit operations/manipulations, and conditional branches. Figure 10 shows that, on average, benchmark traces contain 9.42% of non-collapsable instructions. The `eon` benchmark performs many floating-point operations, hence the large number of non-collapsable instructions.

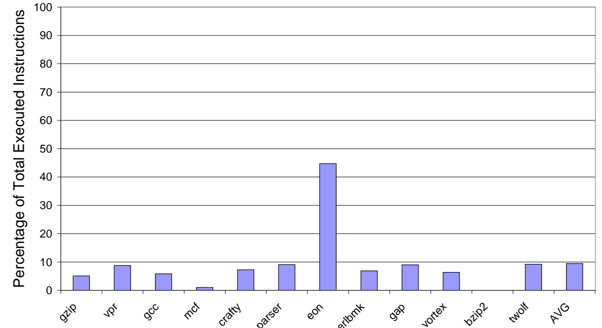


Figure 10: *Percentage of non-collapsable instructions.*

Load instructions. For the moment, *loads* induce 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, 27.7% of executed instructions are load instructions, their irregular occurence in DFGs still enables large DFGs, as shown in Figure 7. Store and branch instructions are not *cuts*, they are exit points of DFGs. Load and store instructions are still considered collapsable : 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 cuts" with true register dependence. By static dependence, we mean a store followed by a load which uses the same register and the same offset for address computation and where the register is not modified between the store and the load. Dynamic store/load dependencies are not detected.

Carries from upper significant bits. Certain combinations of instructions are treated as cuts due to specific carry propagation issues. Consider the example of Figure 11. The result of $(r1 + r2)$ is shifted to the least significant bit, so that the least significant bit of *r6* is added to the most signifi-

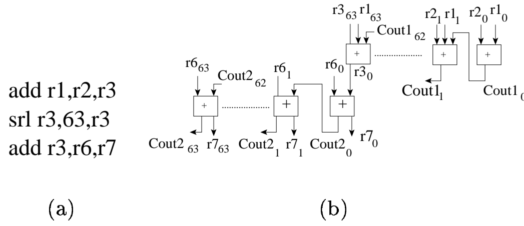


Figure 11: *Cuts because of carries from upper significant bits: (a) assembly code, (b) implementation.*

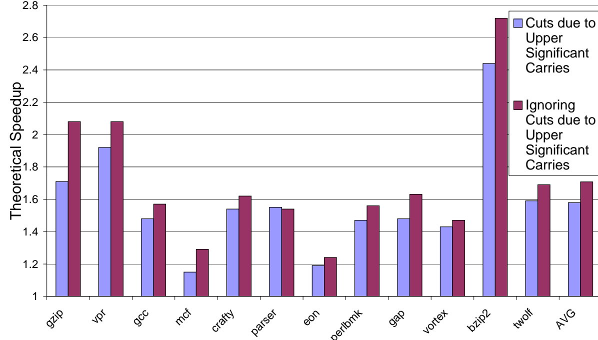


Figure 12: *Effect of relaxing the upper significant carries constraints.*

cant bit of $(r1 + r2)$. Consequently, the addition of *r6* cannot start before $(r1 + r2)$ ends. Therefore, the only way to collapse $(X >> 63) + r6$ with $(r1 + r2)$ is to add another chain of 64 1-bit operators where the output of the most significant 1-bit operator of $(r1+r2)$ is fed into the least significant 1-bit operator of $(X >> 63)+r6$, see Figure 11. In other terms, either the operator chain is larger than the word size and it can accomodate right shifts, or right shifts must be treated as cuts; the same problem occurs whenever a carry comes from upper significant bits. To test the impact of that choice, we have ignored the hardware consequences (cost) of carries from upper significant bits and removed the corresponding cuts. As shown in Figure 12, further performance improvements can be achieved by relaxing this constraints. In all other experiments, we chose to treat such cases as cuts for hardware cost reasons, i.e., we assume a "left-only" carry propagation.

5 Implementation

In this section we discuss implementation issues of functions, then we present an implementation of our mechanism using the rePLay hardware framework [11].

5.1 Hardware implementation of functions

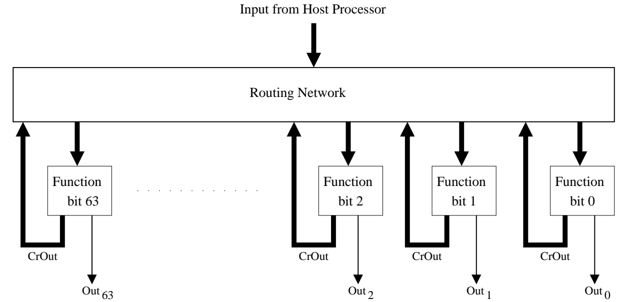


Figure 13: *Implementation of functions.*

Figure 13 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 operators only contain combinatorial 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 [5]. On the other hand, since our approach relies on functions with a significant number of inputs, the number of 1-bit inputs in each operator (10 in our implementation) is larger than the usual 4-input LUTs of many FPGA circuits, resulting in larger logic blocks.

5.2 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. [6][11][12] 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 opti-

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
Function units	8
Branch	2 level Gap predictor, 128 2nd level entries, 14 history wide , 7 cycle BR resolution
Memory Latency	80 cycles
L1 DCache	256kB, 1 cycle
L1 ICache	16kB, 1 cycle
L2 Unified Cache	1MB, 6 cycle

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

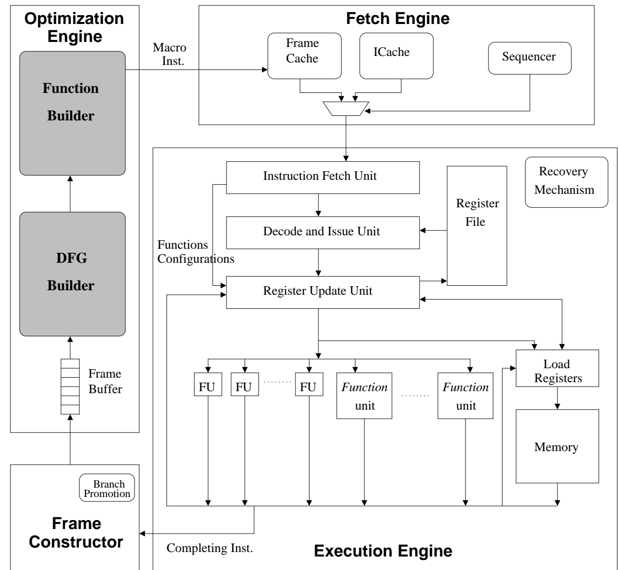


Figure 14: *The core architecture.*

mization engine and the associated *Function* units, in the Simplescalar simulation environment (using the `sim-outorder` model [16]). We assumed a future scaled-up 8-way superscalar processor architecture, see Table 1 for the modified parameters. Figure 14 shows the core processor together with rePLay and the function mechanism which includes several *Function* units. We assumed a maximum of 10 physical inputs for each bit of the *Function* units. Note that our optimization mechanism can be built on top of the optimizations proposed in [6] 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 the performance improvements reported in Section 6 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, described in Section 4, forms the DFG for each frame and transforms the trace of instructions into a trace of functions or *macro-instructions*. These functions are cached into the frame cache and are directtly forwarded to the *Function* units upon a frame cache hit. *Function* units may be configured 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 goes well with our approach. Since each conditional branch is transformed into a 1-bit function, as shown in the example of Figure 2, 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 with *Anticipation* [7].

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 [6] 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.

6 Performance Analysis

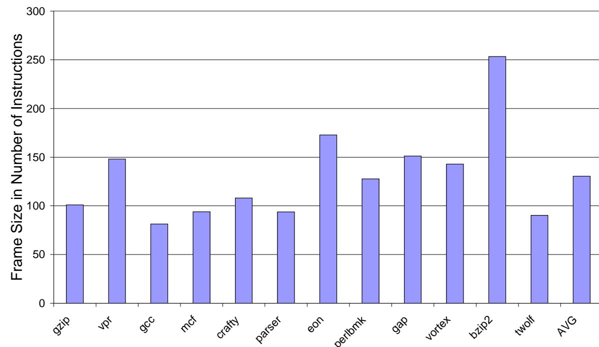


Figure 15: *Average frame size.*

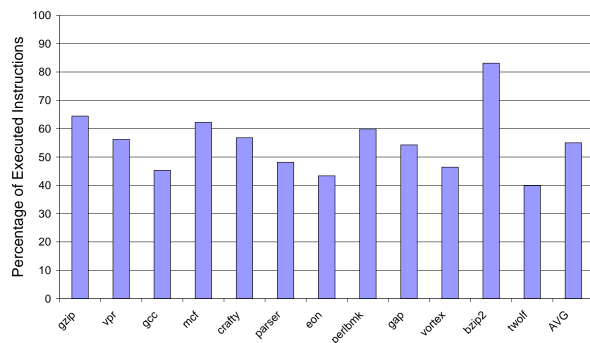


Figure 16: *Dynamic instructions coverage.*

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 130 instructions, see Figure 15. On average, 54.9% of all instructions instances effectively belong to optimized frames, see Figure 16. 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

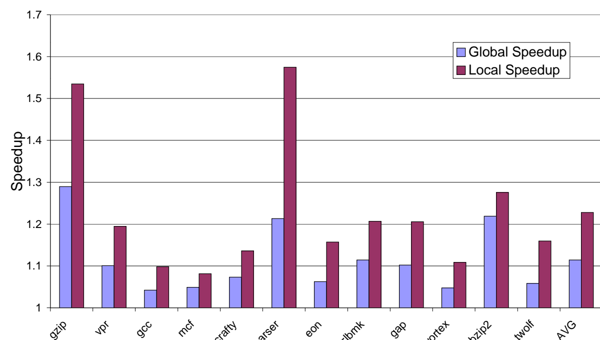
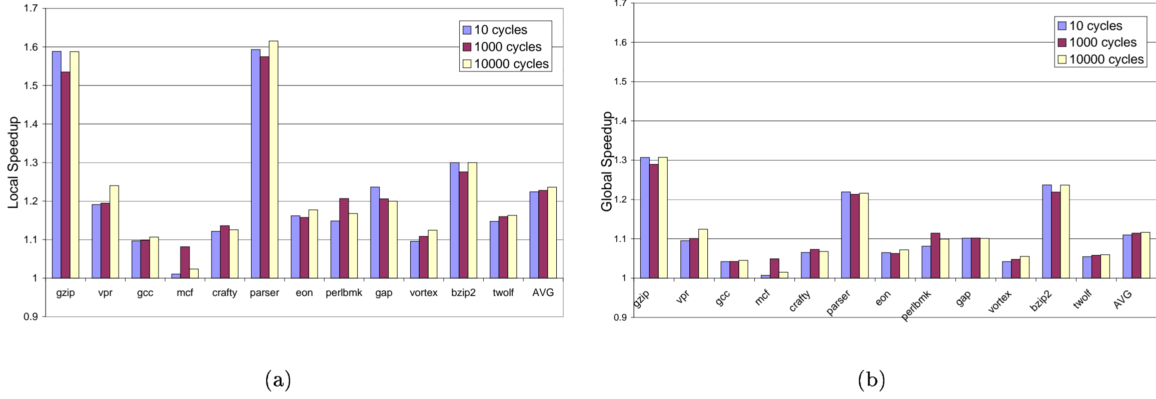


Figure 17: *Global and local speedups.*

scaled-up 8-way superscalar architecture (see Table 1) coupled with rePLay as the baseline configuration; note that the rePLay framework improves the fetch mechanism performance of the superscalar processor [11]. Frames transformed into functions execute 22% faster than the baseline configuration on average, with a maximum of 58% for the `parser` benchmark, as shown in Figure 17. Because of the still limited coverage of the rePLay environment, the global performance improvement is only 11.5% but with strong variations from 0.4% for the `mcf` benchmark to 28% for the `gzip` benchmark. Codes with large sets of sequential and dependent instructions particularly benefit from the mechanism. The low speedups of `mcf` and `eon` benchmarks are mainly due to low branch prediction rate (`mcf`), and the high percentage of non-collapsable instructions (`eon`), see Figure 10.

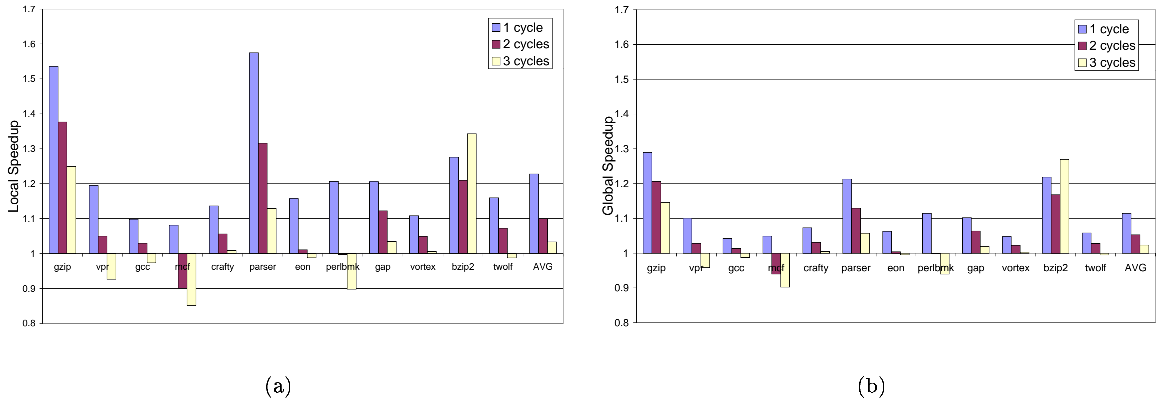
Delay of the optimization engine. As previously discussed, the rePLay framework enables complex dynamic optimizations out of the critical path. Because it is difficult to estimate a priori the exact delay of these optimizations (whether implemented in hardware or software) we varied the optimization delay from 10 cycles to 10000. Figure 18 shows that the optimization engine delay has little or no impact on the speedups. Once optimized, the frames are kept in the frame cache, and frequently reused by the processor, thus lowering the effect of the optimization delay. Also, in some cases, an increased latency of the optimizer prevents useless frames from evicting highly used frames [6], inducing performance improvements in many benchmarks. In all other experiments of this section, we use 1000-cycle delays.

Latency of the *Function* units. Again, because of the unusual nature of the *Function* unit, it is difficult to precisely estimate its latency. Since these chained operators implement multiple but



(a) (b)

Figure 18: *Effect of the optimization engine delay: (a) local speedup, (b) global speedup.*



(a) (b)

Figure 19: *Effect of the* Function *units latency: (a) local speedup, (b) global speedup.*

straightforward carry propagation, they can benefit from the the fast but complex complex carry-propagation schemes that were specifically designed for speeding up FPGA-based carry chains [9], and which are different from the standard high-performance adder carry chains [20]. Therefore, we consider the *Function* unit latency can probably match that of a typical ALU, i.e., 1 cycle. Still, we studied the impact of larger unit latencies on speedup. With a 2-cycle unit latency the average local speedup decreases to 11% and the global speedup to 6%, see Figure 19. Note that the performance of `bzip2` improves with 3-cycles units thanks to higher frame prediction rate. Our optimization starts having an adverse effect for some codes only with a 3-cycle unit latency.

7 Conclusions and Future Work

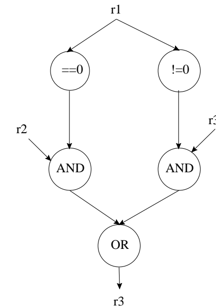


Figure 20: *Conditional instruction* `cmoveq r1,r2,r3`*.*

Extending collapsing with logical guards. One of the difficult issues in forming large traces

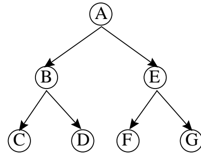


Figure 21: *Control Flow Graph.*

is to deal with conditional branches, and rePLay deals with branches by expressing them as guards associated with traces. However, as the number of conditional branches per trace increases, the probability that a trace will complete its execution decreases. Another approach to conditional branches is to integrate them as guards again in the collapsing process described in Section 2. The size of the hardware implementation of functions would increase, but it would considerably alleviate the trace issue. This latter solution fits well with the scalability-oriented philosophy of translating hardware resources into performance. Let us first illustrate this notion with the simple control flow graph of Figure 20, and then with the more general example of Figure 21. Figure 20 shows the DFG of the conditional instruction `cmoveq r1,r2,r3` which moves `r2` in `r3` if `r1` is equal to zero. Using collapsing, the function becomes the following logical expression: $r3 = (r1 == 0).r2 + (r1! = 0).r3$ which is easily translated into a hardware function using the *Function* unit described in Section 5.1. And this approach can be generalized to any control flow graph. Consider now the control flow graph of Figure 21 where each node represents a basic block. For the sake of simplicity, we will assume each basic block i contains a single function f_i, and ends with a conditional branch Br_i whose logical condition is denoted C_i. Then, the collapsed expression of this tree is equal to function $C_A C_B f_C + C_A \overline{C_B} f_D + \overline{C_A} C_E f_F + \overline{C_A}\ \overline{C_E} f_G$, which can again be mapped to a hardware function using a scaled-up *Function* unit (more inputs and thus larger 1-bit blocks). Therefore the problem of building large traces can be overcome at the cost of additional on-chip space and computation redundancy.

While the hardware cost is high, this approach provides a path for scaling up performance beyond the limitations of branch prediction and the rePLay mechanism.

Conclusions. In this paper, we presented a new approach to improve the performance of sequences of data-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 not excessively complex.

We tested an implementation of this approach on a scaled-up superscalar processor with the rePLay framework, and we observed an average performance improvement of 11.5% , 22% on optimized code sections, with a maximum of 58%.

Besides the branch collapsing direction mentioned above, we are currently investigating extensions of the approach beyond assembly language to higher-level languages, with the prospect of removing many of the current DFG *cuts*.

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] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308, 1996.

[4] Yuan Chou, Pazhani Pillai, Herman Schmit, and John Paul 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] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (CSUR)*, 34(2):171–210, 2002.

[6] 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.

[7] Alexandre Farcy, Olivier Temam, Roger Espasa, and Toni 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.

[8] Dirk Fischer, Jrgen Teich, Michael Thies, and Ralph 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.

[9] Scott Hauck, Matthew Hosler, and Thomas Fry. High performance carry chains for FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(2), April 2000.

[10] Ramadass Nagarajan, Karthikeyan Sankaralingam, Doug Burger, and Stephen W. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, 2001.

[11] Sanjay J. Patel and Steven S. Lumetta. rePLay: A hardware framework for dynamic optimization. *IEEE Transactions on Computers*, 50(6), June 2001.

[12] Sanjay J. Patel, Tony Tung, Satarupa Bose, and Matthew 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.

[13] Sanjay Jeram Patel, Marius Evers, and Yale 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] James Phillips and Stamatis Vassiliadis. High-performance 3-1 interlock collapsing alu's. *IEEE Transactions on Computers*, 43(3), March 1994.

[15] Yiannakis Sazeides, Stamatis Vassiliadis, and James 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.

[16] Gurindar Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39(3), March 1990.

[17] David 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.

[18] Zhi Alex Ye, Andreas Moshovos, Scott Hauck, and Prithviraj 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.

[19] Zhi Alex Ye, Nagaraj Shenoy, and Prithviraj Banerjee. A C compiler for a processor with a reconfigurable functional unit. In *FPGA*, pages 95–100, 2000.

[20] Matthew Ziegler and Mircea 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.