Adding Limited Reconfigurability to Superscalar Processors

Marc Epalza
Signal Processing Institute
marc.epalza@epfl.ch

Paolo Ienne
Processor Architecture Lab
paolo.ienne@epfl.ch

Daniel Mlynek
Signal Processing Institute
daniel.mlynek@epfl.ch
Swiss Federal Institute of Technology (EPFL)
Lausanne, Switzerland

Abstract

When adding reconfigurability to custom hardware, one must take great care that the reduction in speed due to the reconfigurable logic should not cancel out the gains obtained by reconfiguration. These gains are greatest in very specific and computation-intensive applications, and lessen as the applications become more general and heterogeneous. In the case of superscalar processors, this leads to limiting the amount of reconfigurability to precise changes in existing functional units instead of adding a fully configurable functional unit.

We present a detailed study of the modifications necessary in a superscalar processor to allow an FPU to be dynamically reconfigured as several ALUs with a minimal increase in the latency of these functional units. The timing of the FPU's multiplier tree and the decision about reconfiguration are exposed. As there is more than one simple unit involved, this decision is more global than a cycle-by-cycle reconfiguration and must be made for a longer period of time. We discuss possible policies for the dynamic reconfiguration decisions. The results show interesting gains of up to 56% in the best cases, and average gains of 10%, on typical architectures over a wide range of applications.

1. Introduction

General purpose processors are attractive in embedded and consumer applications for their versatility, but the increasing pace of evolution in standards and applications stretches their performance as far as possible. In the mainstream processor domain, any gains in performance allow the use of more complex algorithms or shorten response times and are always desirable. An increase in performance can in part be obtained by the use of fully reconfigurable hardware. However, this reconfigurability comes at the price of a poor logic speed, and can be tolerated only for specific, well chosen applications. In order to apply reconfigurability to general-purpose processors, with no predefined application, we propose to allow only limited reconfiguration to improve performance on all applications. Limiting the amount of reconfiguration and a detailed analysis of the control are necessary to extract gains in a wide variety of applications. This will be illustrated by a modification of some of the functional units of a superscalar processor.

Section 2 will expand on the state of the art, whereas Section 3 will detail our example and address the main design issues. Section 4 explains the methodology followed to obtain the results presented in Section 5. Finally, Section 6 concludes this work and provides insight into possible future directions for limited reconfigurability.

2. Background and Prior Art

Given the limitations of a fixed set of hardware resources, much research has focused on adding some reconfigurability to a general-purpose system. Custom instructions may be used [1], but most of these attempts are based on FPGA technology, using look-up tables that can be written like a memory to build any logic function or system desired. FPGAs are most efficient for code with simple control and large data parallelism (e.g., [2]). Most current methods couple an FPGA fabric with a processor in a single chip, with different methods used to control the reconfiguration, such as VLIW static scheduling [8] or microcode [17]. Several approaches to automatically partition and compile an application into a processor and reconfigurable logic exist [11, 14]. Interesting gains are obtained for selected applications with large parallelism (e.g., [7]).

Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT'04)
1089-795X/04 $ 20.00 IEEE

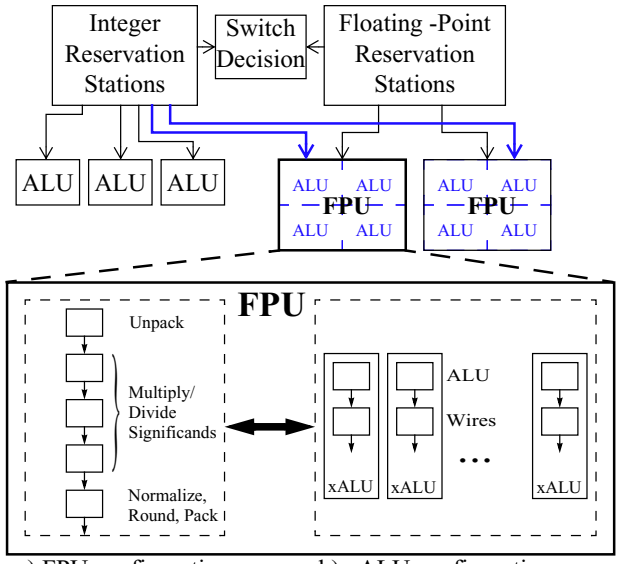

Figure 1. Paths between reservation stations and functional units (top), and reallocation possibilities (bottom). Each FPU can be reallocated as a number of *extra ALUs (xALUs)*. FPU operations have 5 stages, the *xALUs* have 2 stages, all of which must be idle to allow reallocation.

Instead of adding a block of fully reconfigurable logic (such as FPGA technology) and managing the interactions between the two, we propose to consider configuration possibilities as an issue in the design of the processor's functional units. This implies reduced configurability, albeit with a significant gain in speed, which we leverage.

3. Reconfigurable Floating-Point Unit

In a general-purpose processor, only a few parts are viable choices for adding reconfigurability. We focus on the functional units, since we expect the results to be valid for a broad range of applications. Instead of adding a fully configurable unit, whose implementations in an FPGA would result in logic 5 to 10 times slower that the rest of the chip, we propose to add limited configurability to the *floating point unit* (FPU) while maintaining the same clock speed as the non-reconfigurable units. Since many applications do not use the FPU at all, and many others do so intermittently, it is interesting to put some of this hardware to another use.

Our aim is to increase performance while avoiding code changes or having a major impact on timing. The lack of code changes allows our improvements to apply to all existing code and the re-use of all compiler optimizations.

3.1. Concept

Multifunction units, such as the FPUs in the Intel Itanium 2 processor, can execute one of many different instructions each cycle. As shown in Figure 1, we propose to reallocate an FPU, with a latency of 5 cycles as several *extra ALUs* (*xALUs*) with a latency of 2 cycles. These extra ALUs are assumed to perform all the operations normal arithmetic functional units do. Our approach differs from multifunction units because a view over a longer timespan is necessary to offset the idle time between reallocations, as we have to wait for the entire functional unit to be idle before reallocating it. This will be expanded in Section 3.3. We trade a small decrease in speed to obtain some configurability, with the hope that a better matching to individual applications will offset the slightly slower configurable functional units to offer a net gain in performance. We focus on a processor's floating point unit, since it is fairly large, and can often be idle during a program's execution if the current application uses mostly integer code.

Parallel multipliers for fixed point numbers essentially consist of a tree of *Carry-Save Adders* (CSA) that adds all the partial products into two words, with a final *Carry-Propagate Adder* (CPA) for the last addition [10]. The exact structure of the tree may vary to achieve better regularity, essential for good integration.

A division unit can have a similar structure, if a convergence algorithm is used. This would lead to the common implementation of a Mul/Div unit, with a tree structure qualitatively as in Figure 2(a). The number of levels is given by the recursion $h(n) = 1 + h(\lceil 2n/3 \rceil)$, so a 64 input CSA tree has 10 levels [12].

A floating-point Mul/Div unit is essentially a fixed-point Mul/Div unit with some extra logic to unpack the operands, perform Booth recoding (if it is used), normalize the result, and re-pack it into floating-point notation. The presence of a full CPA adder after the compressor tree allows the re-use of the unpack and pack logic to include all floating point operations in the unit. As mentioned above, it is also possible to use the floating point unit for integer multiplication and division, as in the Intel Itanium 2 processor [9].

3.2. Timing Analysis

In an example representing a very aggressive design, the Itanium 2 processor, multiplication (both integer and floating point) takes 4 cycles, whereas ALU operations take a single cycle. Assuming that the floating point operand unpack takes one cycle, and normalize, round and repack take another, the multiplier tree in Figure 2(a) takes the 2 remaining cycles, equivalent to a critical path delay of approximately 15τ in the Figure (10 levels of CSAs plus a single CPA), τ being the delay of a full adder.

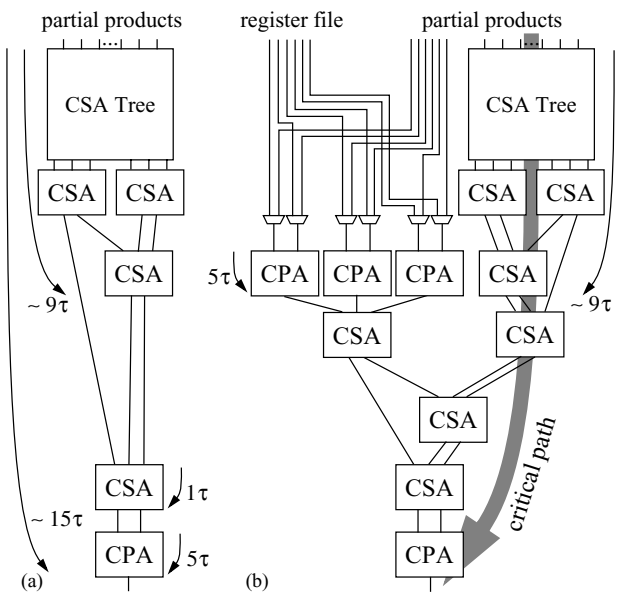

Figure 2. Example of a 64 x 64 multiplier partial product reduction tree. (a) Original tree structure (total delay $\approx 15\tau$). (b) Proposed modification (total delay $\approx 17\tau$). The imbalance in (b) reduces the difference in delay between the two cases, since the CPAs have more time to finish while the values propagate through the CSA tree. The extra delay is due to the 2 extra layers of CSAs. CSAs have a delay of 1τ, CPAs have a delay of about 5τ, 1τ being the delay of a full adder.

Our proposed modification would replace a small number of CSAs with CPAs, in order to use these CPAs when the FPU is idle. This requires unbalancing the tree to give time to the far slower CPAs to finish their work while the other partial products are being summed in the CSA tree, as shown qualitatively in 2(b). The timing consequences of this unbalancing have been studied both theoretically, using technology independent values, and through an implementation of a multiplier in UMC $0.18\mu m$ technology.

3.2.1. Theoretical Analysis. Considering technology independent delays for 64-bit adders [10], a CSA has a delay of 1τ, and a CPA has a delay of around 5τ. As shown in Figure 2(a), the total delay for a balanced compressor tree is 15τ. The unbalanced compressor tree in Figure 2(b) requires at most 2 extra levels of CSAs for a total delay of 17τ, or an increase of about 13%. We do not use the final adder at the bottom of the tree to avoid adding the delay of a multiplexor to the critical path. This also means that it would be possible to perform FP add/subtract operations while using the *xALU* configuration.

3.2.2. Synthesis Results. To confirm the validity of our theoretical analysis, both multipliers in Figure 2 were implemented and synthesized. UMC's $0.18\mu m$ 'typical' technology with the *Artisan* design kit and Synopsys *DesignWare* libraries were used. In each case, only the combinatorial circuit was designed, without any pipeline registers inserted, so the values are directly comparable to the theoretical analysis. The unbalanced tree has been obtained with an algorithm similar to the Three-Greedy Approach [15]. The experimental value for the ratio $\text{Delay}_{CPA}/\text{Delay}_{CSA}$ is 5.4—quite close to the theoretical value. The balanced multiplier in Figure 2(a) reports a critical path delay of 4.1ns, while the unbalanced multiplier in Figure 2(b) has a critical path delay of 4.3ns, with the critical path going through the CSA compressor tree as expected. This is an increase of almost 5%, and is well below the conservative theoretical value estimated above.

As only the FPU's multiplier tree is modified, the impact of our design on power consumption is the sum of three factors: the difference in power between an ALU and an *xALU*, the difference in power between a balanced and an unbalanced multiplier tree, and the difference in execution times. The first is very small, even taking longer wires into account. The difference between the two multiplier trees obtained is an increase of less than 3%. As the results in Section 5 will show, the execution times are almost always reduced by the use of reconfiguration, thus overall power consumption does not increase due to the reconfigurability of the FPU.

The delay of the *xALUs* is mostly dependent on the wires needed to move the operands and results to and from the register file and reservation stations, and the forwarding paths to and from the normal ALUs. Considering that, in deep submicron technology, wires account for about 2/3 of the delays, the placement and routing of the functional units should be done taking these new wires into account. Therefore, we estimate that the delay of the *xALUs* should be about double that of normal ALUs. To be on the conservative side, simulations with a latency of 3 cycles, where about 89% of the delay is due to wires, have also been performed.

In any case, the size and delay of the wires is mostly affected by the number of *xALUs* obtained by reconfiguration of an FPU. A value of 4 in our simulations was chosen because it is the highest value producing interesting gains. An implementation taking cost into account would probably use a lesser value, 2 or 3, at a very small reduction in the speedups obtained. A single *xALU* per FPU still produces gains in the benchmarks that can use it.

3.2.3. Timing Assumptions. The above results can be summarized as timing parameters for our performance sim-

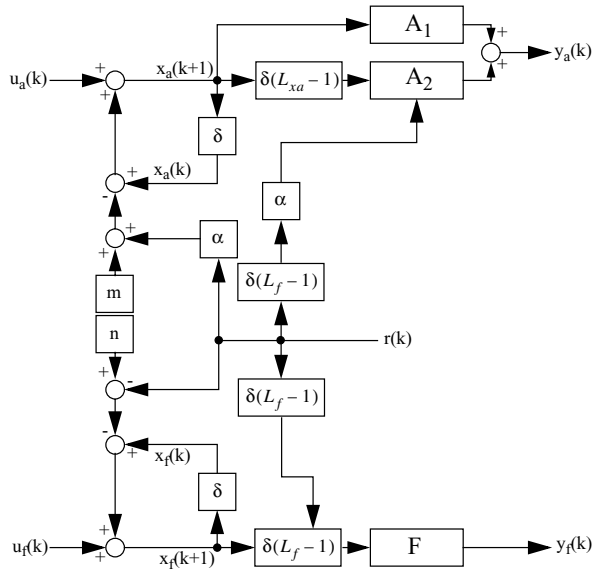

Figure 3. Decision Problem. m and n are the number of ALUs, resp. FPUs. α is the number of *xALUs* per FPU, A_1 and A_2 are the left and right parts of the sum in equation (3), and F is the right part of equation (4). L_{xa} and L_f are the latencies of an *xALU* and an FPU, resp., and $\delta(t)$ represents a delay of t cycles.

ulations: our main timing values will consider the reconfigurable multiplier to take one extra cycle compared to a non reconfigurable multiplier—an increase of 50%, resulting in a total of 3 cycles. This leads to a reconfigurable FPU with a latency of 5. However, given the small measured difference in critical path delay, it might be interesting to consider the possibility of keeping the FPU latency to 4. In all cases, the latency of the *xALUs* is taken as 2, double the latency of a normal ALU, to take wires and multiplexors into account.

3.3. Decision Mechanism

The need for a decision mechanism for the reconfiguration arises from the fact that, unlike when addressing a multifunction unit, decisions for the reconfigurable FPU last for a number of cycles. As results will show, the choice of a decision mechanism can have a considerable impact on the gains obtained by reconfiguration. The complexity of the decision mechanism must also be kept as low as possible, since there are only a few cycles available for each decision.

3.3.1. Problem definition. The decision problem we must solve is the following: Given a set of m ALUs and n FPUs, each reconfigurable as α *xALUs*, and a set of arrivals of *integer* and *floating point* instructions $u_a(k)$ and $u_f(k)$—when all the dependencies of each instruction are resolved—at the discrete times k, $\{k \geq 0\}$, find the set of optimal number of reconfigured FPUs $r(k)$, $\{0 \leq r \leq n\ \forall k\}$ that allows the system to execute all instructions in the shortest time— i.e., that minimizes the finishing time k_{end}. Let L_a, L_f and L_{xa} be the latencies of an ALU, an FPU and an *xALU*, respectively. Let $x_a(k)$ and $x_f(k)$ (defined ≥ 0) be the number of instructions ready and waiting for execution at time k, and finally, let $y_a(k)$ and $y_f(k)$ (also defined ≥ 0) be the number of instructions that commit at time k. The state equations for our system can then be written as:

$$x_a(k+1) = x_a(k) - (m + \alpha \cdot r(k)) + u_a(k) \quad (1)$$
$$x_f(k+1) = x_f(k) - (n - r(k)) + u_f(k) \quad (2)$$

$$y_a(k) = \min[x_a(k), m] + \min\Big\{ \max[x_a(k-(L_{xa}-1)), m], \alpha \cdot r(k-(L_{xa}-1)) \Big\} \quad (3)$$
$$y_f(k) = \min\Big[x_f(k-(L_f-1)), m - r(k-(L_f-1))\Big] \quad (4)$$

These two sets of equations, also shown in Figure 3, model the flow and delay of instructions as they travel through the processor's functional units. Equations (3) and (4) set limits on the number of instructions that can commit, using the minimum of the amount possible under the current configuration and the amount available. From this model, several decision mechanisms can be developed. In all cases, any decision will be limited by the need to wait for a functional unit to be completely idle before reconfiguring it.

3.3.2. Solutions. A naive solution using a *local optimum* to maximize the number of instructions issued at every cycle produced poor results. Another solution, called *balanced*, considers the two equations (1) and (2) as linear functions of $r(k)$ and attempts to balance the number of instructions of each type with the number of appropriate functional units. Adding a weighing factor, we pose $x_a(k) = \lambda \cdot x_f(k)$ to get:

$$x_a(k) - m - \alpha \cdot r = \lambda\Big(x_f(k) - n + r\Big) \quad (5)$$
$$r(k) = \frac{x_a(k) - n + \lambda\Big(n - x_f(k)\Big)}{\lambda + \alpha} \quad (6)$$

The optimal $r(k)$ in this equation is seldom integer, and must thus be rounded to the nearest integer to get the final number of FPUs to reconfigure as *xALUs*. While this

method produces good results in many benchmarks, the calculations in floating point, which might be approximated with integers or fixed point, make an implementation costly, and it is only used as a reference for these benchmarks.

Finally, an experimental approach called *threshold*, derived from control theory, uses simple thresholds with hysteresis to control the changes to the FPU: We first determine the thresholds $T_a(k)$ and $T_f(k)$, and normalize the number of instructions $A(k)$ and $F(k)$ to get $N_a(k)$ and $N_f(k)$. S_{RS} is the size of the reservation stations. As above, m and n are the number of ALUs and FPUs, respectively, $r(k)$ is the number of FPUs reconfigured as *xALUs*, and α is the number of *xALUs* per FPU.

$$
\begin{aligned}
T_a(k) &= m + \alpha \cdot r(k-1) && (7)\\
T_f(k) &= n && (8)\\
N_a(k) &= \lfloor A(k)/(S_{RS}-3) \rfloor && (9)\\
N_f(k) &= \lfloor F(k)/(S_{RS}-4) \rfloor && (10)\\
&\quad \text{with } \{0 \le r(k) \le n \; \forall k\}
\end{aligned}
$$

It is then possible to perform the comparisons and get a decision.

$$
r(k) = \begin{cases}
r(k-1) & \text{if } N_a(k) \le T_a \text{ and } N_f(k) \le T_f \\
r(k-1)+1 & \text{if } N_a(k) > T_a \text{ and } N_f(k) \le T_f \\
r(k-1)-1 & \text{if } N_a(k) \le T_a \text{ and } N_f(k) > T_f \\
n-1 & \text{if } N_f > 0 \text{ and } n = r(k-1)
\end{cases}
$$

The last option is necessary to handle cases where many integer instructions are dependent on very few floating point instructions. Without this check, the threshold to switch an FPU back to floating-point operation would never be reached, and the FP instructions would stall the processor forever. As all the sums can only take a very limited number of values, and the two normalized values are obtained by constant shifts, a very simple and fast implementation can be designed.

4. Experimental Methodology

All the results presented in Section 5 were obtained through the use of the Simplescalar tool set [3], with parameters derived from Section 3.2.3. The models used for the hardware are detailed in Section 4.2. On the software side, the SPEC CPU2000 [5] benchmarks were used for all tests.

4.1. Modifications to Simplescalar

The most accurate simulator in the Simplescalar tool set, sim-outorder, was modified so that a number of FPUs can be turned into several *xALUs*. This required adding functions to create a new resource configuration, and copy the status from one resource pool to another. Several switch decision algorithms were also added to the simulator's main loop, to choose whether and how to change the allocation of resources during program execution. Many statistics about functional unit usage useful to guide our research, notably the tuning of the *threshold* method, were also inserted.

Model	#ALUs (latency)	#FPUs (latency)	#Load/ Store units	#*xALUs* per FPU (latency)	Issue-dispatch-commit widths
Original mainstream	3(1)	2(4)	2	-	4 – 4 – 4
Original top	6(1)	2(4)	4	-	8 – 8 – 8
Baseline mainstream	3(1)	2(4)	4	-	8 – 8 – 8
Baseline top	6(1)	2(4)	5	-	12 – 12 – 8
Dynamic mainstream	3(1)	2(5)	4	4(2)	8 – 8 – 8
Dynamic top	6(1)	2(5)	5	4(2)	12 – 12 – 8
Supertop	10(1)	2(5)	5	-	12 – 12 – 8

Table 1. Processor models. The *baseline mainstream* and *baseline top* models were compared to their *dynamic* counterparts. The *original mainstream* and *original top* models are only shown as references. *Supertop* has 4 more ALUs than *dynamic top* and no reconfiguration.

4.2. Reference Processors and Models

Two different references were used. They are loosely inspired from *mainstream* and *top* server processors available today, and considered representative of the state of the art in general-purpose processors:

Our *mainstream* reference is similar to the IBM Power4 processor (single core) [6], and is close to the average resource configuration of current processors. Each core is a 4-way superscalar processor, and has 2 ALUs, 2 load/store units, one branch unit and 2 FPUs.

Our *top* reference is roughly based on the Intel Itanium 2 processor [9], one of the fastest server processors available today, as measured by SPEC benchmarks [18]. It has 2 ALUs, 4 load/store units that can also perform ALU operations, 3 branch units, and 2 floating point units that also take care of integer multiplication. Although it is a VLIW processor, its resources represent well the most aggressive configuration achievable nowadays. We indicate these processors as *original*.

For a fair comparison, both reference models are also given the same memory access bandwidth and ports as our proposed model (4 or 5 memory ports and a 128-bit wide access to memory), as well as the same issue/dispatch/commit widths, leading to our *baseline mainstream* and *baseline top* models. Finally, the *dynamic mainstream* and *dynamic top* models are obtained by increasing the FPU latency as explained in Section 3.2 and adding dynamic reallocation. *Supertop* is defined as a fully static *top*, with 4 additional

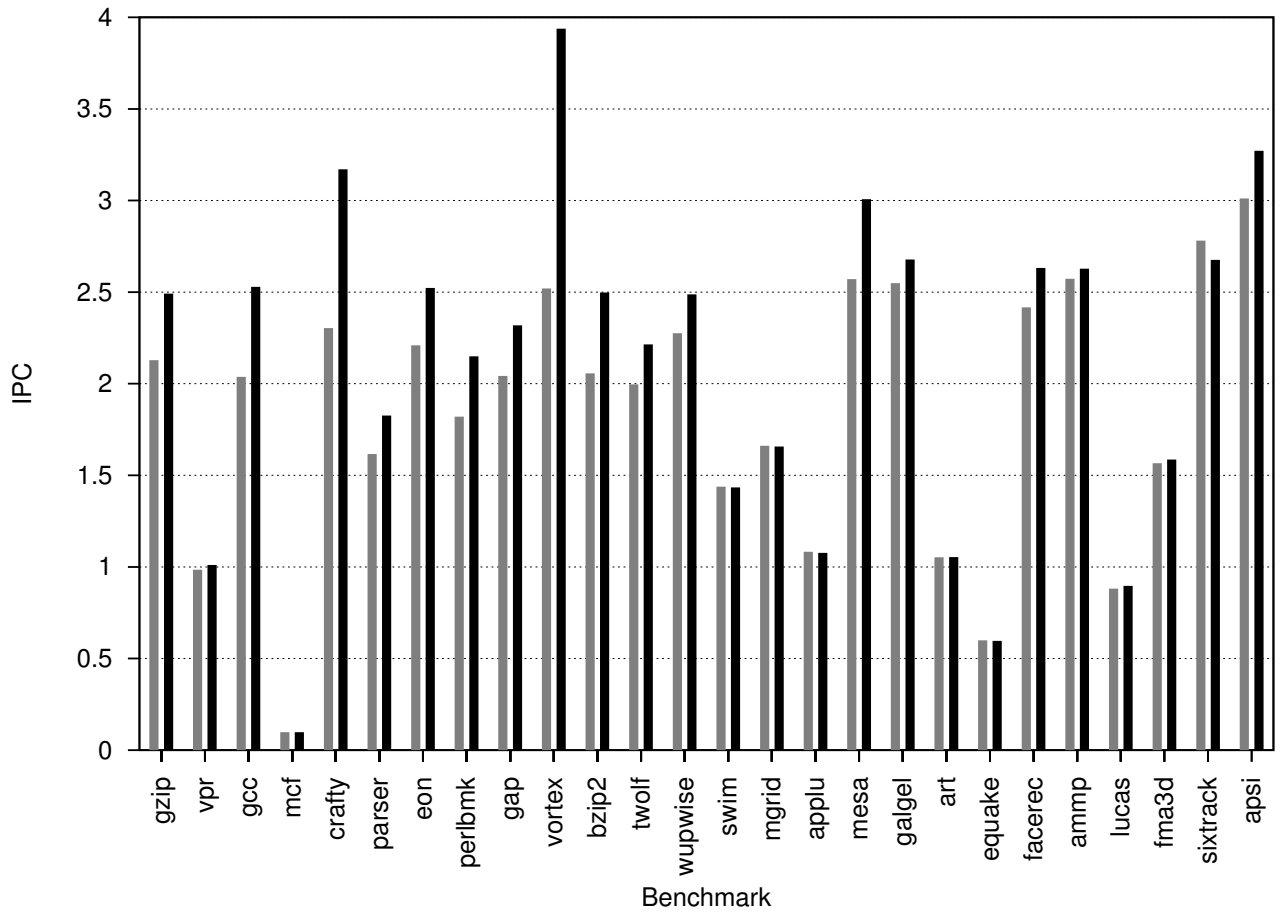

Figure 4. Simulation results for the SPEC benchmarks for the *baseline mainstream* (light) and *dynamic mainstream* (dark) processors. There are large variations in overall IPC, with some significant gains by the *dynamic* model.

ALUs and no reconfiguration, and is used to show the small difference in performance compared to the *dynamic top*. These characteristics are summarized in table 1.

While the *baseline mainstream* is slightly unbalanced, limiting the number of issue slots and load-store units in the *dynamic mainstream* would completely cripple the architecture. Thus, an attempt at finding a fair middle ground is made, where the *dynamic mainstream* cannot issue instructions for all its units in a single cycle.

4.3. SPEC CPU 2000 Benchmarks

All our tests considered the entire set of 26 benchmarks comprising the SPEC CPU2000 suite. The binaries are provided for the DEC Alpha [4] Instruction Set Architecture (ISA) on the Simplescalar WWW site [3], and have been compiled using the 'peak' configuration. The data sets chosen are the reference sets from the SPEC suite.

Due to the length of the benchmarks, the *standard single simulation points* [13] were used for detailed simulation. With these settings, the full SPEC simulation took about 3 weeks on a 2.8GHz Intel Pentium 4 processor. However, the analysis of the influence of the design parameters in Section 5.2 was performed by fastforwarding only 10^9 instructions to keep simulation times reasonable.

The 12 benchmarks from *gzip* to *twolf* are integer, while the 14 benchmarks from *wupwise* to *apsi* are floating point. This slight bias toward floating point benchmarks was maintained in all calculations.

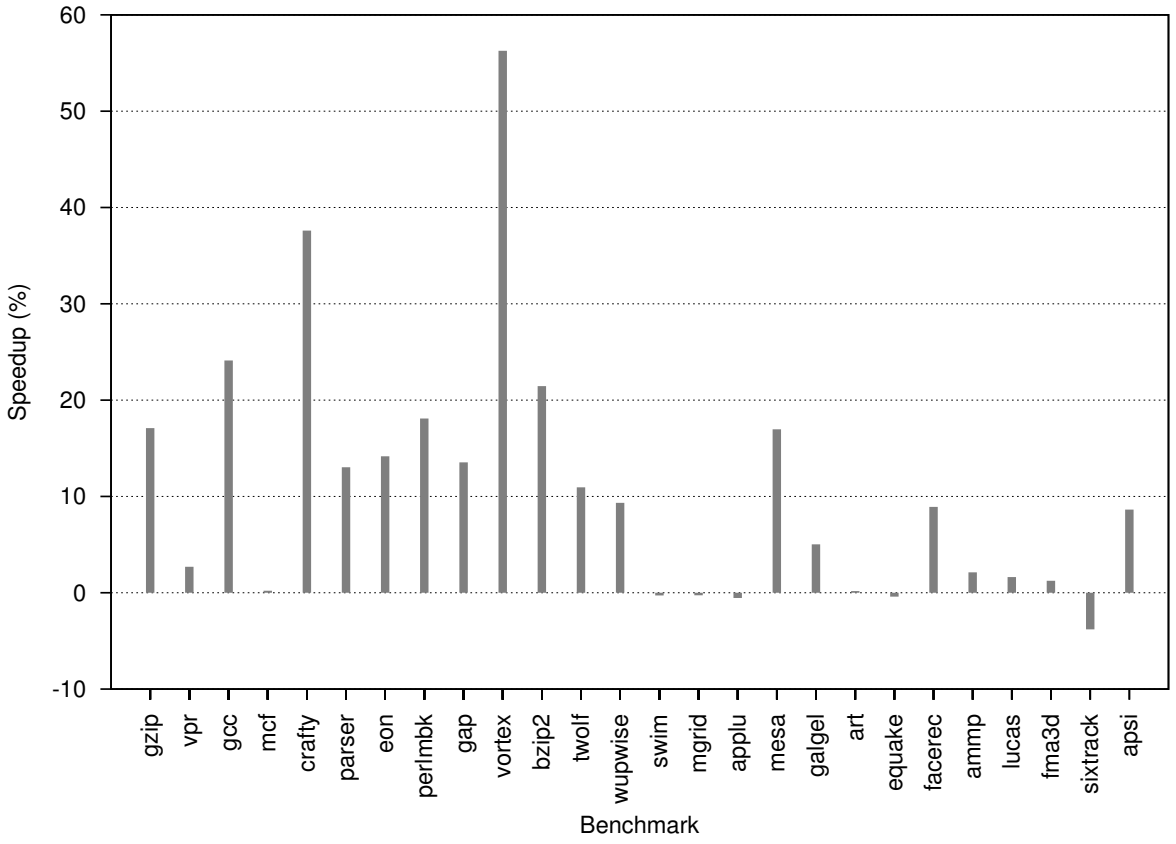

Figure 5. Speedups between the *baseline mainstream* and the *dynamic mainstream* models. The integer benchmarks show universal gains, whereas the FP benchmark results are more varied. All negative speedups are small, less than 0.5% slower than the *baseline*, except for *sixtrack*, with a loss of 3.8%.

5. Results

In this section, the results for all the SPEC CPU2000 benchmarks will be presented, running on both *mainstream* and *top* models. The impact of the different benchmarks and the sensitivity of our proposal to various parameters will be shown. Finally, the impact of the control mechanism will be discussed. Simulations with perfect memories showed almost no difference in the speedups, demonstrating that memory latency has little impact on the performance of our reconfiguration.

5.1. Simulations with the *Threshold* Mechanism

Based on the delay calculations in Section 3.2.1, we use the parameters shown in table 1 to compare the *mainstream* and *top* models. Figure 4 shows the Instructions Per Cycle (IPC) for the *mainstream* model. The best benchmark is *vortex*, since it uses almost only integer instructions and can thus benefit from the *extra ALUs*, although several floating point benchmarks show gains of almost 10%, due to a good mix of integer and floating point instructions. The worst benchmark was *sixtrack*, with a loss of 3.8%, although all the other losses were under 0.5%. This benchmark has little available parallelism and many dependent FP instructions affected by the increase in FPU latency. The average gain for the integer benchmarks was 19%, and 3.5% for the floating point benchmarks. The average over the entire suite was a gain of 10%. The speedups displayed in Figure 5 show a systematic gain, only seldom insignificant, and the few losses in some floating point benchmarks are rather small, with *sixtrack* being a worst-case scenario with 36% of FP multiply operations. The results for *top* found in Figure 6 emphasize the toll from the increase in FPU latency. The best benchmark was again *vortex* with a gain of 11%, while the worst performing benchmark was now *ammp* with a loss of 5.9%. The averages were a gain of 2.5% for integer benchmarks, and a loss of 1.6% for floating point

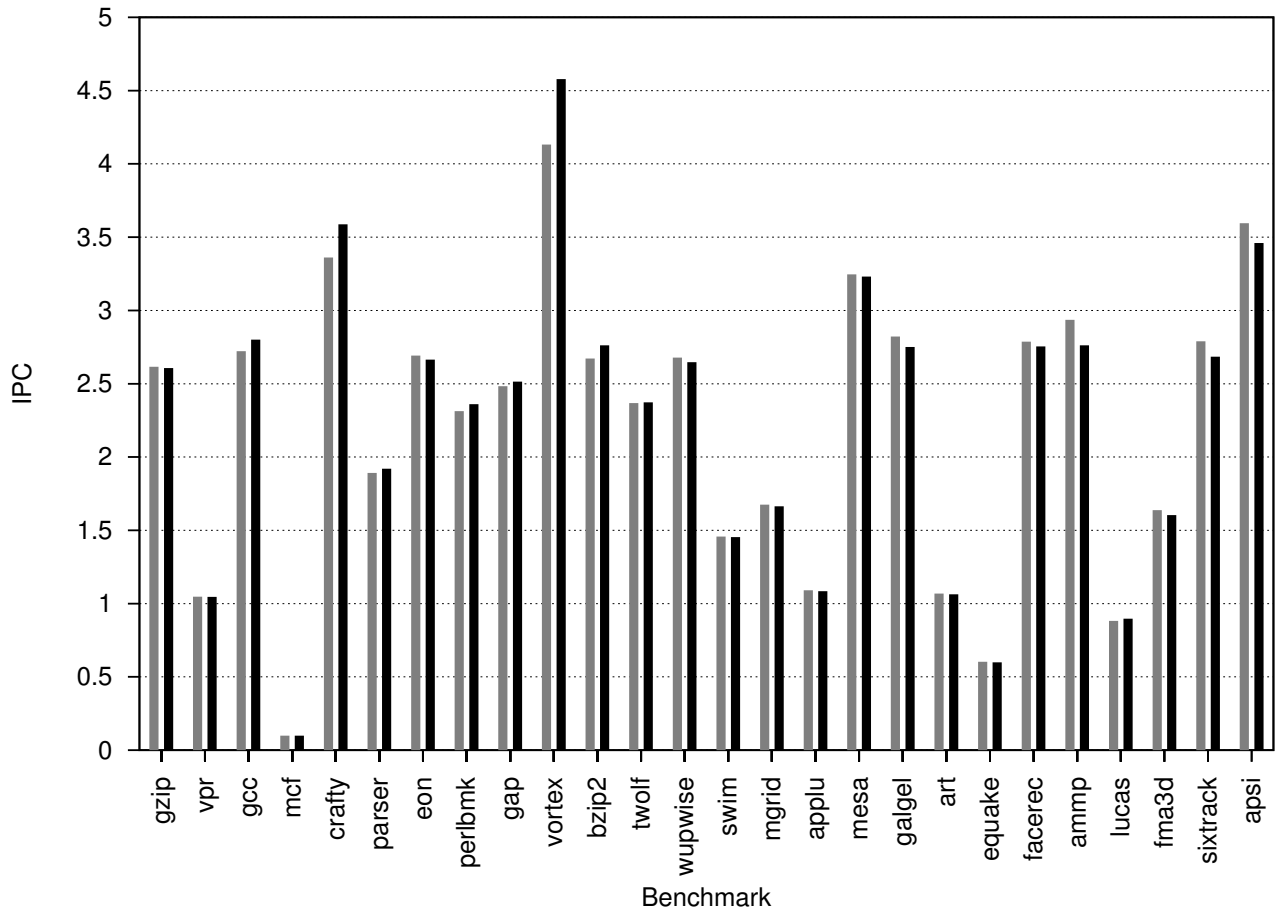

Figure 6. Simulation results for the SPEC benchmarks for the *baseline top* model (light) and *dynamic top* model (dark). The results are usually quite close, with some benchmarks showing larger differences in either direction. The overall average is a tiny gain of 0.24%

benchmarks. The overall average is a barely positive gain of 0.24%. For comparison, the *supertop* model gives an average gain of 1% versus the *baseline top*, at the cost of a larger set of functional units and resources on the die.

5.2. Influence of Design Parameters

As the proposed reconfiguration removes multiplication and floating point resources to better service integer instructions, programs with little or no floating point instructions will benefit most, since they have more integer units available, and they are not affected by the increase in FPU latency. Similarly, as we are increasing the available parallel resources, programs with a greater amount of parallelism benefit more than programs with little parallelism. In the most notable exception, *sixtrack*, the gains obtained by the *xALUs* are eliminated by the increase in FPU latency. As the measured increase in the critical path caused by reconfigurability is rather small, it might be possible, at least in embedded applications, when lowering the frequency—and thus the power, to maintain the FPU latency unchanged at 4, leading to no losses in any application.

The effectiveness of the decision mechanism and the reconfigurability in general is strongly dependent on the instruction types and their scheduling by the processor. In cases like *vortex* where there are almost no instructions for the FPU, the configuration is to use the *xALUs*, and switch only to execute the rare multiplications. In the opposite case, some programs, like *swim* make such heavy use of the FPU that no reconfiguration can ever take place. The middle case, shown in Figure 7, shows a good adaptation of the resources to the instruction types. Increasing the latency of the *xALUs* to 3 reduces the average gain in the *mainstream* model from 10% to 8%, showing that while this latency is

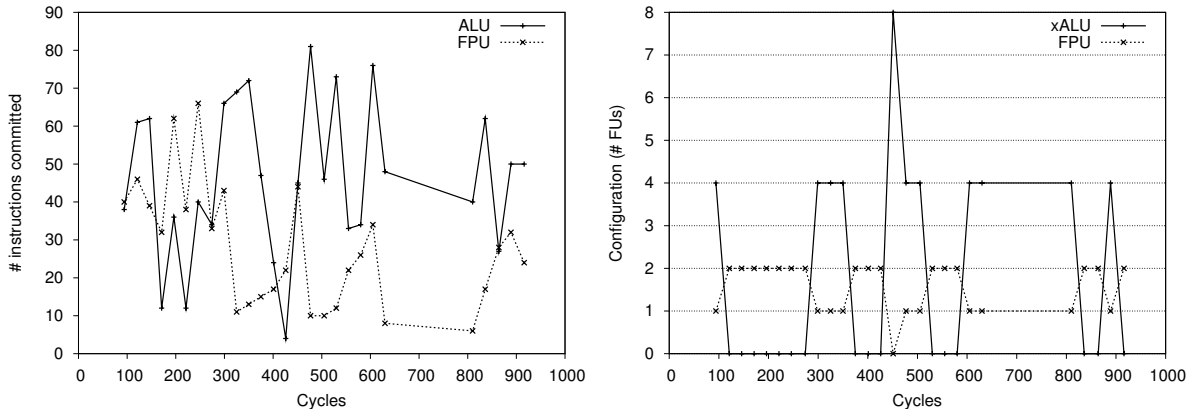

Figure 7. Instruction types (left) and configuration decision (right) for part of the *sixtrack* benchmark. The configuration adapts to the instruction types: when there are few FP instructions, the FPUs are reconfigured as *xALUs*.

somewhat critical to our gain, the benefit does not rely only on these timing assumptions. Further variations of this latency shows that almost all benchmarks make some use of the reconfiguration, with only 2 benchmarks, *swim* and *applu*, never using the *xALUs*. In the benchmarks that make heavy use of the *xALUs*, latencies of 5 or 6 still produce a small gain without affecting the other benchmarks.

Variations of the number of FPUs in the system show little gain in most integer benchmarks from having more than one reconfigurable FPU. This might lead to using only one such FPU, coupled to a normal FPU, which would decrease the penalty in FP benchmarks at a small cost to the gains in integer benchmarks. However, some FP benchmarks see somewhat impressive gains when the number of FPUs is increased to 3 in the *mainstream* models, with *equake* going from a loss of 0.5% to a gain of 7%, and *mesa* slighly increasing its gain from 17% to 22%, with *apsi* also increasing its gain from 9% to 16%. The two latter benchmarks are capable of great parallelism, as further increasing the number of reconfigurable FPUs still increases the gains of reconfiguration, albeit by more modest margins.

Swim and *mgrid* are almost unaffected by FPU latency, and appear to be mainly I/O limited, resulting in no gains from the increased parallelism offered by the reconfigurable FPU.

In our main simulations, the reconfiguration itself was considered to take a single cycle once the decision was made. We have also simulated a greater latency for reconfiguration, during which no instructions may be executed. Increasing this latency had no effect until about 10 cycles, while any value above 50 cycles eliminates all gains, showing the importance of a fast decision mechanism. These simulations also confirmed that *swim* almost completely ignores reconfiguration, since with a huge reconfiguration latency of 1000 cycles, the loss only widens from 0.3% to 1%.

Finally, the impact of the number of *xALUs* obtained by reconfiguring an FPU was examined. There is little gain from using a value higher than 4, except for 2 FP benchmarks, *facerec* and *apsi*. On the other hand, even a single *xALU* produces gains in almost all integer and a few floating point benchmarks, showing the many design choices available with this limited reconfiguration. We expect that any method that increases the amount of parallelism available to the processor, such a Simultaneous Multithreading [16], should increase the value of reconfiguration.

5.3. Impact of the Control Mechanism

The impact of the control mechanism is strongest between the *local optimization* and the two other methods, *threshold* and *balance*. This difference was about 15% on average, and confirms the importance of the decision mechanism. More interestingly, the difference between two algorithms with a large difference in complexity, *balance* and *threshold*, is shown in Figure 8. While *balance*, which is rather complex, is not the theoretical optimum solution, it is fairly close, considering that $r(k)$ must be integer. Hence, the average difference of only 0.14% shows that *threshold* is perfectly appropriate—a decision algorithm that is both simple and effective can be designed.

6. Conclusions and Future Work

We have shown the feasibility of applying reconfigurability to general-purpose processing tasks to obtain interesting gains over a very wide range of applications. Two factors are critical to this gain: first, limiting the reconfigura-

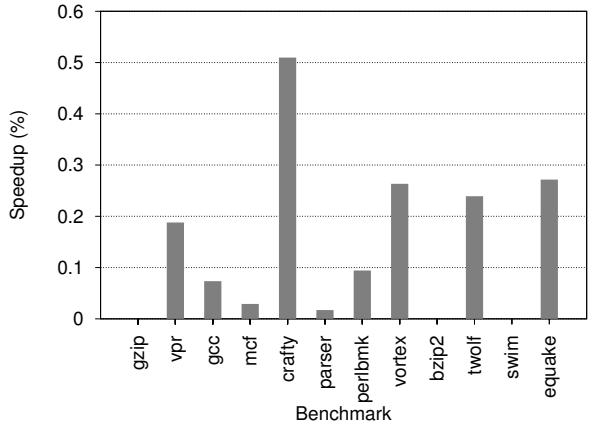

Figure 8. Speedups of *balance* over *threshold*, run on the *mainstream* model. In these benchmarks, *balance* is always the same or better than *threshold*, but with an average gain of only 0.14%. The others (not shown) have negative speedups due to very fast changes in the instruction type distribution and the rounding of $r(k)$.

bility helps minimize the loss in speed due to reconfigurable logic. This requires a detailed analysis of the architecture and applications being considered. Second, great care must be applied to the choice of the configuration decision mechanism, as it can have a considerable impact on the results. The results show that interesting gains can be obtained by adding limited reconfiguration to the FPUs of typical general purpose processors. However, the application to processors with many functional units shows less evident advantage. Power consumption is not significantly affected, and the reconfigurable version might even be better in this regard than a processor with all the static functional units like our *supertop* model because it has less leaky components.

We are currently looking for a near-optimal solution to the problem of the decision mechanism, to be able to quantify the difference between the theoretical gain from reconfigurability and our implementation. The complexity of an implementation of a scheduler with similar functional units with different latencies should also be addressed. We plan to explore the extra parallelism opportunities offered by simultaneous multithreading in the context of general purpose processing, as well as searching for other possible applications of limited reconfigurability.

We would like to thank the reviewers for their helpful and detailed comments.

References

[1] K. Atasu, L. Pozzi, P. Ienne, *Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints*, Proc. of the 40th Design Automation Conference, June 2003.

[2] M. Borgatti, et al., *A Reconfigurable Signal Processing IC with embedded FPGA and Multi-Port Flash Memory*, Proc. of the 40th Design Automation Conference, June 2003.

[3] D. Burger, T. M. Austin, *The Simplescalar Tool Set, Version 2.0*, www.simplescalar.com.

[4] J. H. Edmondson, et al., *Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor*, Digital Technical Journal, 1995.

[5] J. L. Henning, *SPEC CPU2000: Measuring CPU Performance in the New Millennium*, IEEE Computer, July 2000.

[6] J. Kahle, *Power4: A Dual-CPU Processor Chip*, Microprocessor Forum 99, October 1999.

[7] A.L. Rosa, L. Lavagno, C. Passerone, *Hardware/Software Design Space Exploration for a Reconfigurable Processor*, in Proc. of the DATE 2003, March 2003, pp. 570-575.

[8] A. Lodi, M. Toma, F. Campi, A. Cappelli, R. Canegallo, R. Guerrieri, *A VLIW Processor with Reconfigurable Instruction Set for Embedded Applications*, in ISSCC Digest of Technical Papers, February 2003, pp.250-251.

[9] C. McNairy, D. Soltis, *Itanium 2 Processor Microarchitecture*, IEEE Micro, March 2003.

[10] A. R. Omondi, *Computer Arithmetic Systems: Algorithms, Architecture and Implementations*, Prentice Hall, 1994.

[11] E. M. Panainte, K. Bertels, S. Vassiliadis, *Compiling for the Molen Programming Paradigm*, in Proc. of the 13th Int'l Conference on Field-Programmable Logic and Applications (FPL), vol 2778, Springer-Verlag Lecture Notes in Computer Science (LNCS), September 2003, pp. 900-910.

[12] B. Parhami, *Computer Arithmetic Algorithms and Hardware Designs*, Oxford University Press, 2000.

[13] E. Perelman, G. Hamerly, B. Calder, *Picking Statistically Valid and Early Simulation Points*, Int'l Conference on Parallel Architectures and Compilation Techniques, September 2003.

[14] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, E. Deprettere, *System Design using Kahn Process Networks: The Compaan/Laura Approach*, Proc. of the DATE 2004, February 2004.

[15] P. Stelling, C. Martel, V. Oklobdzija, R. Ravi, *Optimal Circuits for Parallel Multipliers*, IEEE Transactions on Computers, March 1998.

[16] D. Tullsen, S. Eggers, H. Levy, *Simultaneous Multithreading: Maximizing On-Chip Parallelism*, Proc. of the 22nd Annual Int'l Symposium on Computer Architecture, June 1995.

[17] S. Vassiliadis, S. Wong, S. Cotofana, *The MOLEN rm-coded Processor*, in Proc. of the 11th Int'l Conference on Field-Programmable Logic and Applications (FPL), vol 2147, Springer-Verlag Lecture Notes in Computer Science (LNCS), August 2001, pp. 275-285.

[18] SPEC CPU 2000 Results published by SPEC, http://www.spec.org/cpu2000/results/cpu2000.html.