Dynamic Reallocation of Functional Units in Superscalar Processors

Marc Epalza[1], Paolo Ienne[2], and Daniel Mlynek[1]

[1] Laboratoire de Traitement des Signaux 3,
[2] Laboratoire d'Architecture des Processeurs,
Swiss Institute of Technology Lausanne (EPFL), 1015 Ecublens, Switzerland
{marc.epalza, paolo.ienne, daniel.mlynek}@epfl.ch

Abstract. In the context of general-purpose processing, an increasing number of diverse functional units are added to cover a wide spectrum of applications. However, it is still possible to design custom logic adapted to a particular application that will perform far better than a processor. In an attempt to give it some adaptability, adding some reconfigurability can help improve performance. We propose to extend the possibilities of complex multifunction units by dynamically reallocating existing complex functional units as multiple simpler units. The fact that more than one simple unit is involved in the "reconfiguration" process implies that the decision is more global and needs to be taken for a longer period of time. We show that in typical superscalar architectures, there are no major impediments to implementing such a decision scheme, and that on a specific reallocation opportunity we can achieve speedups of up to 56% over a mainstream superscalar processor and practically no losses.

1 Introduction

In general purpose processors, the quest for ever higher performance leads to many trade-offs, since one aims to achieve the best average performance on a variety of tasks essentially unknown to the designer. Many methods to extract even more parallelism, such as speculative execution or *Very Long Instruction Word* (VLIW) compiler technologies are complex and achieve diminishing returns, since the resources available to the processor are fixed. Attempts to make the processor adaptable to the program it is currently executing, through the use of reconfigurable logic, have provided mixed results. We propose to introduce some adaptability without using slow reconfigurable logic. To this end, we focus on the large multi-function units present in a superscalar processor. As an example, we expose a modification of a superscalar processor's *floating point functional units* (FPU) to allow some adaptation to the current workload.

Section 2 will lay out the constraints of the field and existing methods to achieve high performance. Next, section 3 will present our proposal and its impact on processor design. Our test methodology and reference processors will be exposed in section 4, with simulation results shown in section 5. Section 6 will bring our conclusions, the limitations of our approach, and our future directions of study.

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 185–198, 2004.

2 Background and Prior Art

2.1 Parallelism

The way to higher performance in general purpose processors is through forms of parallelism, especially by trying to execute as many instructions as possible at the same time. The theoretical limits in the parallelism offered by different programs are far higher than those achieved in reality by current processors, for a variety of reasons. In any case, the available hardware resources are fixed by the processor's designer, and cannot be tailored to a particular application. They are chosen to get the best average performance. Superscalar processors, executing many instructions out of order every cycle, extract as much parallelism as possible during the execution of a program. This leads to complex designs, but these optimizations don't require changes to the software. In an attempt to soften the restrictions of fixed hardware resources, configurable hardware has been examined.

2.2 Reconfigurable Functional Units

Given the limitations of a fixed set of hardware resources, much research has focused on adding some reconfigurability to a general-purpose system, usually based on FPGA technology. FPGAs are most efficient for code with simple control and large data parallelism (e.g., [2]).

One can distinguish three different approaches, each bringing closer integration with the processor, and thus more generality, at the expense of performance. The first and second couple an FPGA and a normal processor, and distribute the computing tasks according to what each can do best, the difference being whether to integrate the FPGA onto the processor chip or not. There is little automation possible, and selection and coding for the FPGA must be done by hand, including the needed communication and synchronization with the processor. With well chosen applications, the gains in performance can be of several orders of magnitude [15]. In a single-chip solution, some automation is possible, usually with a smaller increase in performance than if optimizations are performed by hand (e.g., [13]).

The last, most tightly coupled solution is to define the configurable logic as simply an extra *functional unit* (FU) of the processor. This reconfigurable functional unit can hold several instructions or sequences of instructions, that can be provided by a special compiler, and loaded by the processor when needed. Attempts to automate the process exist (e.g., [1], [16]), with gains similar to the second solution above (e.g., [5]). In each of these cases, the approach is to couple an existing FPGA-style block with a processor, in a more or less tightly coupled way.

We propose to consider configuration possibilities as an issue in the design of the processor's functional units, instead of adding a block of existing fully reconfigurable logic (such as FPGA technology) and trying to have the two cooperate. This implies a reduction in the configurability available, albeit with a significant gain in speed, which we hope to leverage.

2.3 Binary Compatibility

The issue of binary compatibility, ensuring that all code written for previous versions of a processor family will work on the newest model, is a complex one. However, it limits the innovation that can be implemented in the processor, since no completely novel approach may be used. As a solution to this problem, dynamic binary translation has been proposed. It aims to transform code for one architecture into another in real-time during the execution of the program. Several research projects exist [14], with one commercial implementation [8].

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 achievements. Preserving the general timing will avoid breaking or severely limiting the performance of existing programs not suited to our modifications.

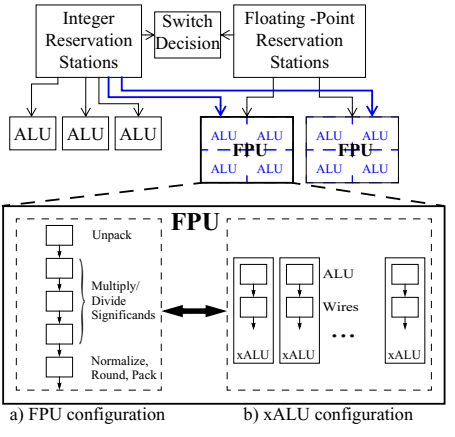

Fig. 1. Paths between reservation stations and functional units (top), and reallocation possibilities (bottom). Each FPU can be reallocated as a number of *extra ALUs (xALUs)*. FPU operations have 5 stages, thus the FPU must be idle for 5 cycles before reallocation is possible. Likewise, the *xALUs* have 2 stages, and must all be idle for 2 cycles at the same time to allow reallocation.

3 Proposed Modification

Studies on the ideal mix and functionality of functional units in a superscalar processor have been performed [7]. These studies show that good gains can be obtained by increasing the number of identical functional units, as well as the types of instructions these units can execute. We are interested in looking for ways to reconfigure expensive functional units to perform different operations. Given the

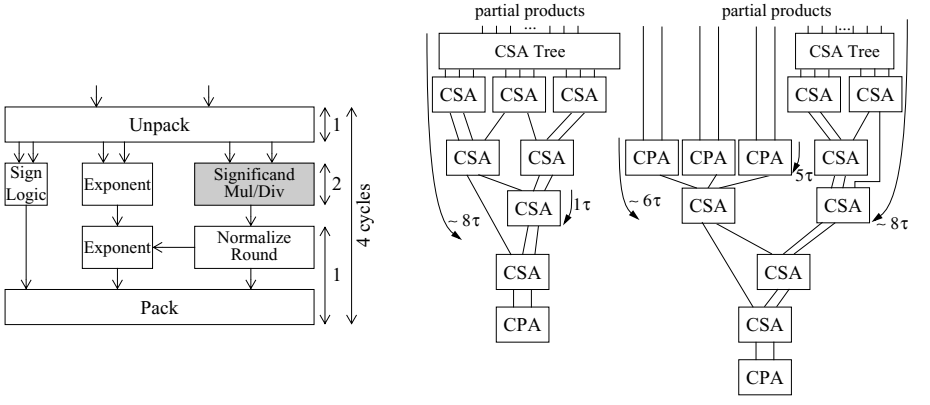

Fig. 2. Left: Structure of a floating point multiply/divide unit, with assumed cycle counts. Center and Right: Example of a 64 bit multiplier partial product reduction tree. Center: Original Wallace tree structure (total delay 14τ). Right: Proposed modification (total delay 15τ). CSAs have a delay of 1τ, CPAs have a delay of about 5τ. For clarity, the multiplexers from the Register file to the CPAs for the *xALU* configuration are not shown.

speed disadvantage of fully programmable units, which are 5 to 10 times slower than a dedicated custom logic in the same technology, we restrict ourselves to very limited changes, while maintaining speeds close to non-configurable logic.

3.1 Basic 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 (figure 1a) as several *extra ALUs (xALUs)* with a latency of 2 (figure 1b). These *extra ALUs* are assumed to perform all the operations normal Arithmetic Functional Units do. Our approach differs from multifunction units since, due to these latencies, the reconfiguration decision cannot be taken on a cycle-by-cycle basis, but with a view to the next several dozen cycles. This longer view is necessary to offset the idle time before reallocation, as we have to wait for the entire functional unit to be idle before reallocating it. We trade a small decrease in speed to obtain some configurability, with the hope that adapting to 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. Simply adding extra ALUs would further increase power consumption and area, with little impact on the results (section 5).

3.2 Standard Arithmetic Units

Current fast multipliers for fixed point numbers can be built from 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 [11], with the tree structure qualitatively as in figure 2 (center). Each CSA or CPA block might need an inverter to allow subtraction. The CSA tree has $\lceil \log_{3/2}(64) \rceil = 9$ levels.

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, as shown in figure 2 (left). The presence of a full CPA adder allows the re-use of the unpack and pack logic to include all floating point operations in the unit. It is also possible to use the floating point unit for integer multiplication and division, as in the Intel Itanium 2 processor [9].

3.3 Dynamic Functional Units

We propose to use the adders in an FPU as a number of *xALUs*, with characteristics similar to normal ALUs. As a CSA cannot be used to perform a complete addition, several CSAs in the tree could be replaced by CPA adders as in figure 2 (right) with only a minimal impact on the overall critical path, area and power consumption. This figure shows the proposed modifications to the reduction tree, which affect only the steering of the data, not the logic performed on it. The CPAs directly receive some of the partial products while the other partial products go through the CSA tree to allow time for the far slower CPAs to finish execution, resulting in only a small extra delay due to the unbalancing of the tree. This requires some extra logic: to handle logic operations other than add/subtract, to bring the operands for the extra instructions that will be executed, to bypass the floating point logic, and to switch between the two different modes of execution.

3.4 Effects on Functional Unit Latencies

Our reference for instruction latencies is the Intel Itanium 2 processor, one of the fastest (and certainly the largest) existing processor [17]. This processor has a latency of 1 for all ALU operations, and a latency of 4 for all FP operations and Integer Mul/Div operations. These latencies are considered here representative of current 64-bit processors, and the functional units are fully pipelined.

In deep sub-micron technology, such as 0.13μm, wires account for about 2/3 of the delays, and the differences between 0.13μm and 0.09μm are not so important in this regard. The increase in wiring to reach the *xALUs* is estimated at about double that needed for normal ALUs. Thus, if a normal ALU has a

latency of 1 cycle, split as 1/3 gates and 2/3 wires, doubling the wires gives a *xALU* latency of 5/3. Taking the multiplexers to select the adders in the FPU into account, a conservative estimate for the latency of all *extra ALU* units is to double the latency of normal ALUs, for a latency of 2. As confirming this timing would require designing the entire functional core of a superscalar processor, a complex task beyond our means, simulations with a very conservative latency of 3, where about 89% of the delay is in the wires, have also been performed. Additionally, some of the bypass paths necessary to keep the pipeline as full as possible, and counted in the above calculations, are likely to already be present in the multiplier's tree linking the *xALUs* together. This also means that the overhead is less than that of simply adding extra ALUs to the processor.

The latency of the entire FPU being 4, we consider that the unpack stage takes one cycle, the multiplier tree takes 2 cycles, and the normalization and pack take the last cycle (figure 2 left). The replacement of some of the CSA adders by CPA adders will increase the total delay of the multiplier tree. From [10], considering that a CSA has a delay of 1τ, a delay of 5τ for a 64-bit CPA can be derived. The total delay for a 64-bit CSA tree with 9 levels (see section 3.2) and the final CPA, is thus $9\tau + 5\tau = 14\tau$ (figure 2 center). We assume this delay represents 2 cycles (figure 2 left), as both real processor data [9] and arithmetic considerations [11] suggest. As shown in figure 2 (right), implementing our modifications on the FPU to embed 3 CPAs in the compressor tree would increase its delay to $8\tau + 2\tau + 5\tau = 15\tau$ plus the delay a multiplexer in front of each CPA (figure 2 right). To be on the conservative side, and since functional unit latencies must be integral, we have assumed the total delay of the modified tree to be 21τ, equivalent to 3 cycles, an increase of 50%, or one cycle, for a total delay of 5 cycles in the functional unit. This adds a margin of 6τ, almost 40%, that is, many layers of logic, to the timing of the FPUs CSA tree. In any case, the partial products reduction tree is a logarithmic tree which can be easily unbalanced as needed to hide the delay of the CPAs, and so the inaccuracy due to the delays of the multiplexors and the bypass paths should not be significant overall.

Since the reconfiguration is achieved by switching the inputs of a few multiplexers, it takes only a single cycle, in addition to having to wait for the functional units to be idle, with no changes to the pipeline except the activation of the forwarding paths discussed above. The routing of the processor core must be redone to take the new data paths into account, but this kind of work must be done for newer technologies in any case. These numbers are summarized in table 1.

3.5 Switch Decision Mechanism

Given the possibility of changing an FPU into a number of xALUs, the issue of deciding when to perform this change, and when to change back, is posed. Since this decision cannot be taken every cycle because it is a global decision affecting several functional units (figure 1), an algorithm to adapt the resources to the code running at a given moment is needed. The basis for the decision is the type

of instructions in the reservation stations. This gives a measure of the type of instructions the processor can expect to be executing a few cycles later. In the simplest case, the number of instructions of each type are then compared to the number of available functional units of the same type to make a decision. A *switch* is decided when the difference between the proportion of instructions of a type in the reservation stations and the resources of that type becomes too large. In the relatively common case that an instruction type should appear very infrequently, such as an integer program with very few multiplications, the algorithm above will not trigger a *switch*, since the threshold is not reached by a single instruction. In this case, we must detect that an instruction cannot be executed due to the absence of the correct resource type, and force a *switch*, regardless of the contents of the reservation stations. In all cases, a *switch decision* must wait until the functional unit(s) it wants to reallocate are completely idle, in which case it takes only a single cycle. It would be possible to *switch* while the FPU is still finishing the last calculation, during the normalization/pack stage, but this would greatly increase the complexity of the control path without a great effect on performance, through the pipelining of the *switch* logic and extra complexity in the pipeline.

3.6 Additional Considerations

The act of *switching* one or more FPUs into a number of *xALUs* increases the pressure on the memory system, as well as providing the need for extra issue, dispatch and commit width. Though the memory bandwidth remains the same, a higher number of Load/Store units are required to avoid stalling the processor due to many memory requests. In our simulations, 4 such units (as in the Itanium 2) were a good balance between performance and complexity. The widest issue rate in current processors is 8 instructions per cycle [17]. A larger issue rate increased the gains of dynamic reconfiguration, but only slightly. Thus, the issue and dispatch widths were kept at 8. The commit width need not be as large as the issue/dispatch width, since the average number of instructions committed per cycle is lower than the maximum. In our simulations, the highest average IPC was slightly below 4 (*vortex*), leading to a commit width of 8 to avoid limiting performance, as the simulator used requires it to be a power of 2, although a value of 4 could be considered.

4 Experimental Methodology

All the results presented in section 5 were obtained through the use of the Simplescalar tool set [3]. The models used for the hardware are detailed in section 4.2. On the software side, the SPEC CPU2000 [6] benchmarks were used for all tests.

Table 1. Processor model resources. The *baseline mainstream* and *baseline top* processors were compared to their *dynamic* counterparts in all simulations, with *original mainstream* and *original top* shown as references. *supertop* is equivalent to *dynamic top* with 4 additional ALUs and no reconfiguration.

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

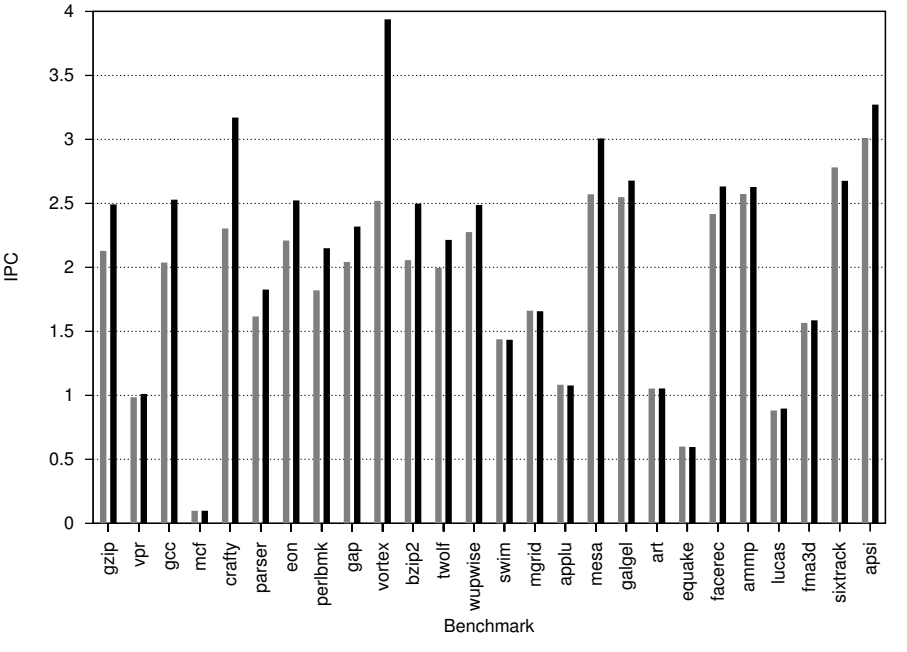

Fig. 3. Simulation results for the SPEC benchmarks for the *baseline mainstream* (light) and *dynamic mainstream* (dark) processors. There are large variations in the overall IPC, with some significant gains by the *dynamic* model.

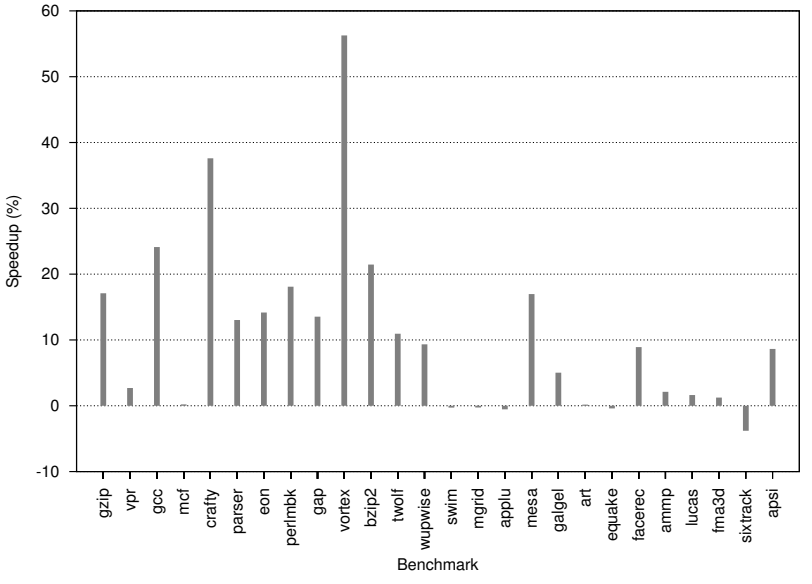

Fig. 4. Speedups between the *baseline mainstream* and the *dynamic mainstream* models. The integer benchmarks show universal gains, whereas the FP benchmark results are more varied. Except for *sixtrack*, all negative speedups are very small, less than 1% slower than the *baseline*.

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*. The *switch decision algorithm* was also added to the simulator's main loop, to choose whether and how to change the allocation of resources during program execution.

4.2 Reference Processors and Models

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

Our *mainstream* reference is similar to the IBM Power4 processor (a single core), 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 loosely based on the Intel Itanium 2 processor, one of the fastest server processors available today, as measured by SPEC benchmarks. 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.

For a fair comparison, both reference models are given the same memory access bandwidth and ports as our proposed model (4 or 5 load/store units and a 128-bit wide access to memory), as well as the same issue/dispatch/commit widths, giving us our *baseline mainstream* and *baseline top* models. Although these models are somewhat unbalanced, not increasing the number of load/store units would cripple the *dynamic* models, which are obtained by increasing the FPU latency as explained in section 3.4 and adding dynamic reallocation. *Supertop* is defined as a fully static top, with 4 additional 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.

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. given the length of the full simulations, early *Simpoints* [12] were used to provide statistically significant results for the *mainstream* model, detailed in figures 3 and 4. Due to time constraints, and since they are only intended to show the limits of reallocation, the *top* and *supertop* models were simulated skipping a smaller number of instructions than *Simpoint* suggests. Although the individual results may vary, the average over the 26 benchmarks is similar to that obtained using *early simpoints*, and sufficient to show a trend of diminishing returns.

5 Results

5.1 Performance Results

Figure 3 shows the results of our simulations for the *mainstream* model, using the configurations in table 1, lines 3 and 5. The speedups when using perfect memories, not shown, show little difference with those presented here, demonstrating that reasonable memory latencies have little effect on the gains made by dynamic reallocation. The best performing benchmark was *vortex*, with a gain of 56%, since it uses many independent ALU operations and very few FP instructions, thus being able to make good use of the $xALUs$, and the worst was *sixtrack*, with a loss of 3.8%, which is mostly composed of FP add and multiply, and is thus strongly 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 overall average for the entire suite was a gain of a little more than 10%. For clarity, the corresponding speedups for the entire set of benchmarks are shown in figure 4. There is a systematic gain, only seldom insignificant, and the rare losses in heavily FP-oriented benchmarks are rather small, with the exception of *sixtrack*.

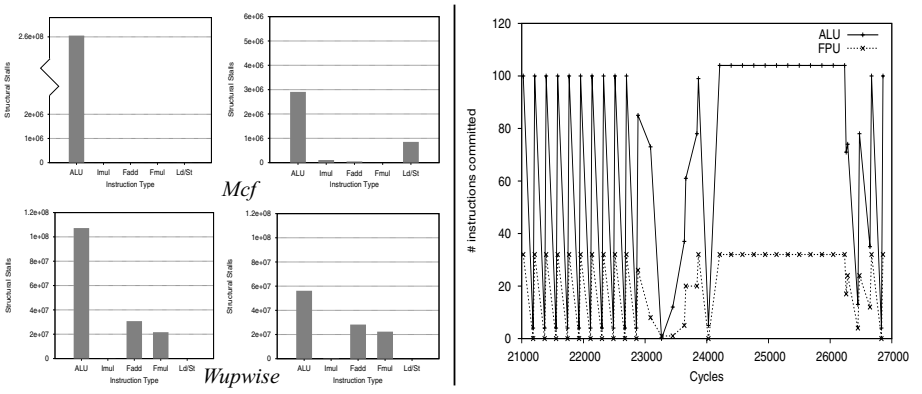

Fig. 5. Left: Structural stalls for *mcf* (top) and *wupwise* (bottom). The left side is the *mainstream baseline* case, the right side is with *dynamic reallocation*. *Mcf* is limited by ALU instructions, and shows a large reduction in ALU stalls. *Wupwise* sees little change in stalls, and thus cannot benefit from reallocation. Right: Instruction types for *galgel*. As there is no region with few FP instructions and many ALU requests, the allocation decision is to have no *xALUs*, resulting in lower performance.

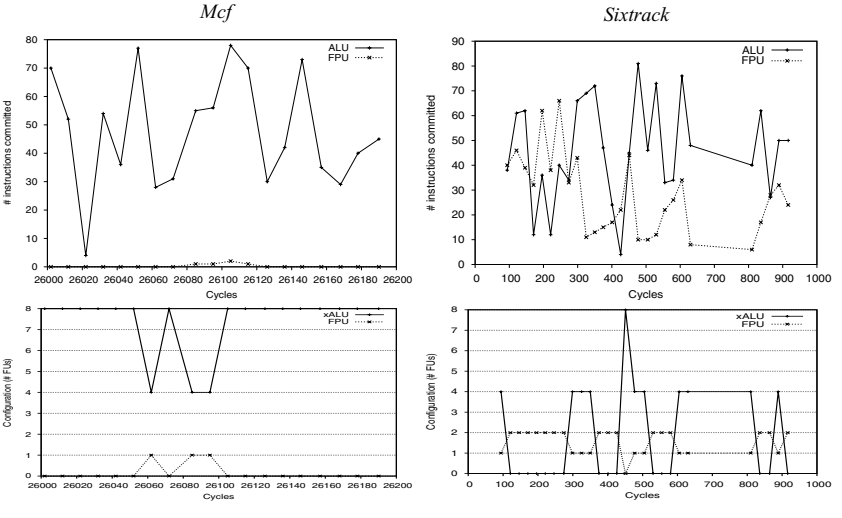

Fig. 6. Instruction types (top) and resource allocation (bottom) for *mcf* (left) and *sixtrack* (right). For *mcf*, as there are almost no FPU instructions, the configuration is always to use 8 *xALUs*. When an FPU instruction arrives, the FPU is switched to execute it, and then immediately switches back. In the case of *sixtrack*, the allocation of the FPU's resources adapts to the instruction types: when there are few FPU instructions, the units will be reallocated as *xALUs*.

The results for the *top* model, described by lines 4 and 6 in table 1, show a reduction in the gains obtained, due to far less usage of the *xALUs*, as there are already 6 ALUs in the processor. Again, memory latency did not significantly affect the speedups. The average gains were 3.7% for integer benchmarks, and 1.5% for floating-point, giving a total average gain of 2.5%. For comparison, the *Supertop* model gives an average gain of 3.1% versus the *baseline top*, at the cost of a larger set of functional units and resources on the die. If the *xALUs* latency is increased to 3, the results show a reduction in the average gain from 10% to 7%, and in the maximum gain from 56% to 35%. Thus, although this delay is somehow critical to our gain, the benefit of our system does not fully rely on these timing assumptions. Losses are not affected, since these benchmarks rarely use the *xALUs*, if ever.

5.2 Influence of Instruction Types

The large differences in speedups for the different benchmarks can be explained by looking at the instruction types used in these benchmarks. We shall use three benchmarks to illustrate this point: *mcf*, *wupwise* and *galgel*. The following graphs show good examples of the different behaviors reallocation produces. However, these are not necessarily representative of the overall benchmark results. Figure 5 (left) shows the number of structural stalls—i.e., the number of instructions of each type which had all operands ready, but couldn't execute due to a lack of functional unit, for the first two benchmarks with the *mainstream* model. The former, *mcf*, is limited here almost only by ALU instructions in addition to memory accesses, and thus benefits greatly from our proposal, since both FPUs get reallocated into many *xALUs*, switching back regularly to service the FP operations. This behavior is shown in figure 6 (left). The limitation by the Load/Store units appears because all ALU instructions that were previously waiting for a functional unit have been executed by one of the *xALUs*, and the memory accesses that had time to execute in the *baseline* case now stall the processor while waiting for the Load/Store units, which are now far less numerous than the ALUs. On the other hand, *wupwise* uses a fairly diverse mix of instruction types, with a heavy emphasis on floating-point add and multiply/divide instructions. The *switching mechanism* is constantly reallocating the functional units to try to match the instruction mix at each moment in time. In this case, the *extra ALUs* available at some moments cannot compensate for the slowdown of the FPUs' mul/div units and the delays in switching between the two. To illustrate this, a short trace of the instruction types for galgel is shown in figure 5 (right). The corresponding *switch decision*, not shown, is to never use the *xALUs*, leading to a loss in performance due to a longer latency in the FPU.

5.3 Switching Dynamics

For the resource reallocation to work, the *switch mechanism* must configure the hardware to make the best use of the configurable resources. Figure 6 (right) shows a short trace from the *sixtrack* benchmark, taken after approximately

10^9 instructions. Figure 6 (top right) displays the number of instructions committed from the ALUs and the FPUs, while figure 6 (bottom right) shows the configuration of the FPU over the same period of time.

The pattern shown is one of the startup loops in the application, and repeats regularly around the instruction count shown. At around 200 cycles, there are more FPU instructions than ALU ones, and the switching mechanism does not allocate any *xALUs*. However, at 300 cycles, the situation reverses, and one FPU is converted into 4 *xALUs*. A sharp spike in ALU instructions coupled with a sharp drop in FP instructions at 450 cycles will cause both FPUs to be reallocated as 8 *xALUs* for a brief moment, before resuming FP functions. A long period of relative stability, between 650 and 850 cycles leads to a unchanging configuration.

6 Conclusions and Future Work

We have proposed a method to gain some hardware adaptability to the code running on a general-purpose processor that does not sacrifice the speed of the configurable unit or compromise binary compatibility. This technique is distinctive in requiring the logic of the superscalar processor to make more global decisions than it normally does. The conditions for the simulations have been derived from real data measured from 0.13μm technology. The results show the use of a dynamic FPU is quite interesting in the case of processors with a modest number of ALUs, and that naturally the interest declines with a large number of ALUs already in the processor. Our idea, based on giving the processor more possibilities for parallelism, should be seen as an example of the possibilities in superscalar processors that can be exploited by multi-cycle reallocation decisions. When superscalar processors will enter the embedded System-on-Chip world, the common use of domain-specific instructions or coprocessors for these applications will increase the opportunities for similar forms of reconfiguration.

We intend to apply control theory to the decision mechanism, in order to better tailor the resources to the application. Simulations on a SMT processor are expected to produce interesting results, due to the extra parallelism exposed by the multiple threads. We also envision to research the possibility of using software hints in the code to guide resource reallocation. While this would maintain backward binary compatibility, it will require a recompilation and some analysis of the code to produce better gains. In a similar vein, it might also be possible to apply this method to VLIW processors, in which case the resource allocation would simply be another information generated by the compiler.

Acknowledgment. We would like to thank the anonymous reviewers for their insightful 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. S. Hauck, T. W. Fry, M. M. Hosler, J. P. Kao, *The Chimaera Reconfigurable Functional Unit*, IEEE Symposium on Field-Programmable Custom Computing Machines, 1997.
6. J. L. Henning, *SPEC CPU2000: Measuring CPU Performance in the New Millennium*, IEEE COMPUTER, July 2000.
7. S. Jourdan, P. Sainrat, D. Litaize, *Exploring Configurations of Functional Units in Out-of-Order Superscalar Processors*, Proc. 22nd Annual Int'l Symposium on Computer Architecture, June 1995.
8. A. Klaiber, *The technology behind Crusoe processors*, Transmeta Corporation, Jan. 2000.
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. B. Parhami, *Computer Arithmetic Algorithms and Hardware Designs*, Oxford University Press, 2000.
12. E. Perelman, G. Hamerly, B. Calder, *Picking Statistically Valid and Early Simulation Points*, International Conference on Parallel Architectures and Compilation Techniques, September 2003.
13. R. Razdan, M. D. Smith, *A High-Performance Microarchitecture with Hardware-Programmable Functional Units*, Proc. of MICRO-27, Nov. 1994.
14. G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, S. Amarasinghe, *Dynamic Native Optimization of Interpreters*, IVME 03, June 2003.
15. R. D. Wittig, *OneChip: An FPGA Processor With Reconfigurable Logic*, IEEE Symposium on FPGAs for Custom Computing Machines, 1995.
16. Z. A. Ye, N. Shenoy, P. Banerjee, *A C Compiler for a Processor with a Reconfigurable Functional Unit*, ACM Int'l Symposium on Field Programmable Gate Arrays, 2000.
17. In-Stat/MDR Workstation and Server Processor Chart, `http://www.mdronline.com/mpr/cw/cw_wks.html`