Factors Influencing the Performance of a CPU-RFU Hybrid Architecture

Girish Venkataramani, Suraj Sudhir, Mihai Budiu and Seth Copen Goldstein

Carnegie Mellon University
Pittsburgh PA 15213
{girish,ssudhir,mihaib,seth}@cs.cmu.edu

Abstract. Closely coupling a reconfigurable fabric with a conventional processor has been shown to successfully improve the system performance. However, today's superscalar processors are both complex and adept at extracting Instruction Level Parallelism (ILP), which introduces many complex issues to the design of a hybrid CPU-RFU system. This paper examines the design of a superscalar processor augmented with a closely-coupled reconfigurable fabric. It identifies architectural and compiler issues that affect the performance of the overall system. Previous efforts at combining a processor core with a reconfigurable fabric are examined in the light of these issues. We also present simulation results that emphasize the impact of these factors.

1 Introduction

The continued scaling of the minimum feature size of CMOS technology provides the chip real estate necessary to place a sizable reconfigurable fabric on the same die as a processor. Reconfigurable computing devices have been shown to achieve substantial performance improvements over conventional processors on some computational kernels [4]. These benefits arise from hardware customization, which avoids the mismatch between the basic requirements of the algorithms and the architectures of the processors. Previous research has shown that closely coupling conventional processor cores with such reconfigurable fabrics is the preferred method of deploying these devices, since these devices can be ill-suited in executing entire applications due to space limitations, and inefficiency in implementing some operations (e.g. floating-point arithmetic).

Modern superscalar architectures present powerful techniques to exploit Instruction Level Parallelism (ILP) from a dynamic instruction stream. These processors allow multiple instruction issue and out-of-order execution. Augmenting these processors with a reconfigurable fabric may offer many opportunities to significantly boost performance, since several instructions, from the CPU and Reconfigurable Functional Unit (RFU), can be executed concurrently.

We focus on automatically mapping general-purpose applications to a hybrid architecture, and identify critical factors that can affect this system's performance. These factors fall into two broad categories – the CPU-RFU interface design, and compiler support. The former is heavily influenced by the CPU architecture. For example, the ability to concurrently execute CPU instructions and RFU configurations can better exploit the features of superscalar architectures. However, if they are both allowed to access a shared address space, then memory consistency becomes an issue. Compiler support determines what kind of code regions (instruction sequences, loops, whole functions) will form *RFUops*. An RFUop is an operation that executes on the fabric.

This paper identifies and describes a number of factors that represent potential bottlenecks to performance. Resolving these problems gives direction to the design of the hybrid system. In Section 2, we present the architectural issues that are relevant in hybrid machine design. In Section 3, we examine the necessary compiler support required to boost performance. In Section 4, we describe our tool-chain. Our experimental framework is presented in Section 5, where we also analyze our results. In Section 6, we discuss previous hybrid designs in the light of the issues presented in this paper.

2 The Hybrid Architecture

Many choices face the designer of a hybrid processor. For every feature under consideration, the costs and benefits must be weighed. The costs may come in the form of additional hardware support, runtime overheads, and complexity. The primary benefit of adding a new capability to the reconfigurable fabric is enlarging the selection space of possible RFUops that can be placed on the fabric, leading to increased performance.

2.1 ISA Additions

An RFUop may be represented as a series of instructions that trigger the RFUop's execution, allow access to data, and write back results. This can either be done with a single instruction or with multiple instructions. We have chosen to use multiple instructions, since it gives us more flexibility to experiment with the design. We define `inputrfu` to be an instruction that will transfer the RFUop's input data from the CPU's register file to the fabric. The `startrfu` instruction triggers the execution of the RFUop. This instruction must specify the RFUop's identification number to select the RFUop. The `outputrfu` instruction transfers the results of the RFUop back to the register file.

The number of registers that can be specified with a single `inputrfu` (or `outputrfu`) instruction is limited by the instruction encoding width. Multiple `inputrfu` (or `outputrfu`) instructions are used if necessary.

2.2 Effect on the Superscalar Pipeline

Superscalar architectures are characterized by a multiple instruction issue, out-of-order core, with the ability to execute speculatively. In our hybrid architecture, the RFUop instructions go through the pipeline just like the rest of the instructions. When an RFUop is executing, the `outputrfu` instruction does not commit until the RFUop completes. Since, instructions are always committed in order, no instructions beyond the `outputrfu` instructions can commit. This causes a structural hazard, since the reorder buffer will soon fill up.

The effect of this hazard can be ameliorated by re-ordering instructions such that the `outputrfu` instruction appears as late in the instruction stream as possible (e.g. just before the first instruction that is data dependent on the RFUop). Similarly, the `inputrfu` and `startrfu` instructions can be moved up the instruction stream, in order to start RFUop execution as soon as possible. The instructions appearing between the `startrfu` and `outputrfu` instructions can execute and commit, thereby reducing the pipeline congestion.

2.3 Memory Interface

The most suitable type of memory access interface depends intimately on the kind of applications that are targeted. Some previous research has focused exclusively on accelerating streaming multimedia and image processing applications on the reconfigurable fabric. In such cases, it is often useful to have Direct Memory Access (DMA), since access patterns are regular and predictable [10, 5].

Our research focuses on general-purpose C programs. In such applications, it is difficult to gather RFUops of reasonable sizes without including load/stores. Since both the CPU and the RFU can access a share address space, the memory interface needs to synchronize all accesses to ensure consistency.

In superscalar processors, memory accesses are dispatched from a structure such as a load-store queue (LSQ). The LSQ checks for dependencies between its entries, thereby ensuring memory consistency. When the CPU executes concurrently with an RFUop, there is a possibility that memory accesses from the RFUop arrive at the LSQ after a memory reference instruction that follows the RFUop in the program order. In this situation, memory inconsistency could arise if both the memory references access the same memory blocks. We can imagine three different scenarios:

- No Memory Accesses by RFUop: In the trivial case, the memory instructions following the RFUop can be issued freely.
- CPU and RFU access disjoint memory addresses: These accesses can also be permitted.
- CPU and RFU access common memory addresses: To maintain memory consistency, there has to be a mechanism to serialize such accesses.

While ensuring memory consistency, it is important to limit the negative impact on performance that this can potentially cause. Thus we define a protocol which allows memory access from the first two classes to execute in parallel with the RFUop while those in the third class are not issued until the RFUop completes. The accesses from the first and second cases can be scheduled between the `startrfu` and `outputrfu` instructions, while the memory accesses in the third case must appear after the `outputrfu` instruction. However, implementing this policy effectively is difficult since completely disambiguating memory accesses is hard even with the best pointer analyses [7].

An orthogonal issue is the sequence of accesses that originate from the fabric alone. Due to the placement of the RFUop on the fabric, it may be the case that memory references don't arrive at the LSQ in program order, although they are issued in order. A memory reference from later in the program may be placed closer to the fabric's external interfaces, and thus may arrive at the LSQ before an earlier reference.

2.4 Concurrent Active Configurations

With superscalar processors, it is possible to issue multiple `startrfu` instructions concurrently. However, support for multiple active configurations introduces other complexities - contentions for fabric resources, like external I/O pins, and space availability. When the RFUops contain memory references, memory consistency is an issue. If all the references in two RFUops cannot be disambiguated, then they may have to be serialized.

Not allowing concurrent execution of RFUops, on the other hand, could severely restrict the pipeline. Starting with the second RFUop all instructions will stagnate in the fetch queue, waiting for the first RFUop to complete. Compiler directed instruction re-ordering can help in this case if instructions that are not dependent on the second RFUop are scheduled before it.

3 Compiler Support

In this paper we are interested in automatic compilation of high-level languages to hybrid systems. The compiler support is thus essential for the successful utilization of an RFU. In addition to standard optimizations, the key component of such a compiler is to partition the code between the RFU and the CPU and to generate the configurations for the RFU, as well as the interface code between the two. We are addressing here just the automatic application partitioning.

The primary objective of code partitioning is to identify code fragments that can be accelerated by execution on the fabric. A number of architectural constraints may restrict the types of the RFUops that are eventually extracted. Some operations may be too costly to implement in reconfigurable hardware (e.g., floating point computations), or may require complex non-computational processing (e.g., procedure calls, exceptions, system calls).

The potential search space for RFUop selection is huge. Hence, we need to develop a systematic approach to finding the code segments that are most likely to form RFUops. The selected segments should have sufficient parallelism, and should be sufficiently large to amortize the overhead cost of the RFUop invocation. From a structural point of view, the following types of code fragments can be selected for implementation as an RFUop:

- Short sequences of dependent instructions which do not fully utilize the processor datapath. These can be collapsed into *custom instructions*, which are implemented very efficiently on the RFU.
- Basic blocks: are attractive since they are relatively easy to analyze. However, the average size of basic blocks is small, resulting in small RFUops which often cannot amortize the RFU invocation overhead.
- Loops: are a natural candidate for classical parallelizing compilers. Similarly, they are attractive as RFUops, as loops can concentrate a large percentage of the program running time, and can benefit from pipelining.
- Hyperblocks: are contiguous groups of basic blocks, having a single control-flow entry point, but possibly multiple exits. Hyperblocks offer the potential for increased ILP through the use of predicated execution, which removes control dependences by speculating execution along many control-flow paths.
- Single Entry—Single Exit (SESE) Regions: hyperblocks implemented as RFUops raise the additional complication of returning not only data from the RFU, but also control-flow information on RFUop execution termination, which requires a more complicated hardware-software interface. SESE regions do not suffer from this problem. Moreover, natural loops often form SESE regions.
- Whole functions: are particular cases of SESE regions. Prior research has not considered this partitioning strategy, as RFUops were mostly constrained by the available hardware resources. The exponential increase of hardware resources however will make this factor less important, and will thus make whole function selection an attractive solution. The main advantage of this approach is simplicity: there are few alternatives to consider, no new interfaces have to be introduced into the program, functions represent "autonomous" regions of computation, with relatively few inputs and outputs, and program ILP profiles roughly correspond to function divisions.

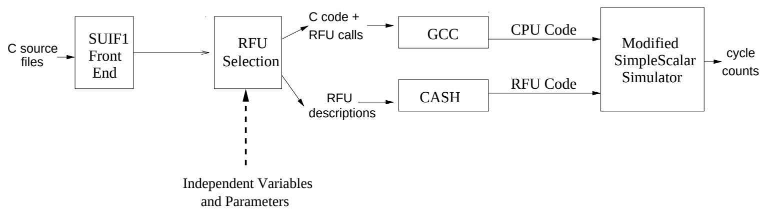


Fig. 1. The Compiler-Simulator tool chain of the system.

- Larger application fragments. The best reported speed-ups for hybrid systems involve hand-partitioned applications, where the portions with high parallelism are implemented on the RFU, and the rest of the computation is handled by the CPU. However, automatic parallelism extraction from large application fragments is still an elusive goal of compiler research.

4 Tool Chain

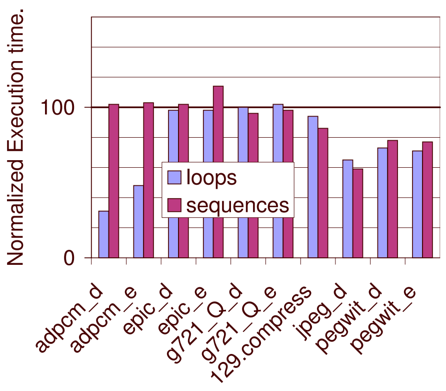
In this section we describe the tool chain we have used to conduct our experiments (Figure 1). It has two parts: Compilation, and CPU-RFU simulation. There are two major phases in compilation - RFUop selection and configuration code generation for the RFU.

The RFUop selection pass is influenced by a number of independent parameters and constraints. These parameters may be classified under two broad categories—architectural and policy-driven. The architectural constraints define the boundaries of the RFU and the CPU-RFU interface. Currently, we support the parameterization of the following architectural constraints—whether memory operations are permissible from the fabric, and the maximum number of permissible RFUop inputs and outputs. The policy-driven parameters aid in refining the potential RFUop search space within the given architectural constraints. These variables define the structural profile of the selected RFUops. We define the following policies:

- Select only natural loops
- Select only short sequences of dependent instruction sequences; i.e., custom instructions
- Restrict the minimum size of RFUops (in terms of the number of instructions)

Further, we assume that there are two hard architectural constraints that are not parameterizable—RFUops must have only one control exit, and RFUops must be self-contained, i.e., RFUops cannot invoke other RFUops or other functions meant to execute on the CPU.

We can view these constraint categories as filters that help in selecting the best RFUops. In addition to these two filters, it is essential to have another filter that diagnoses the goodness of the RFUops selected by the first two filters. This filter will use heuristics like RFUop latency, area and average-ILP estimates to determine RFUop goodness. Without this, the selection process blindly generates too many RFUops. This increases total execution time for two reasons. First, it increases the RFUop invocation overhead. Second, and most importantly, it reduces the effectiveness of traditional compiler optimizations by creating regions of the program which are opaque to the compiler; effectively disallowing certain instruction sequences from being considered



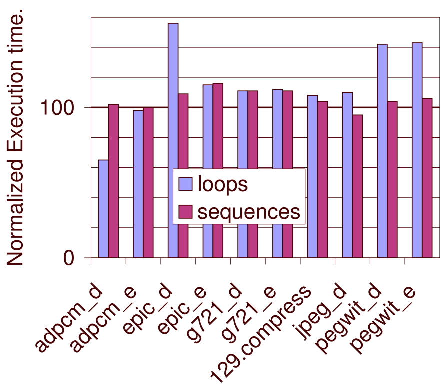


Fig. 2. *Performance comparison between a (single-issue, in-order) MIPS core with the same core augmented with an RFU*

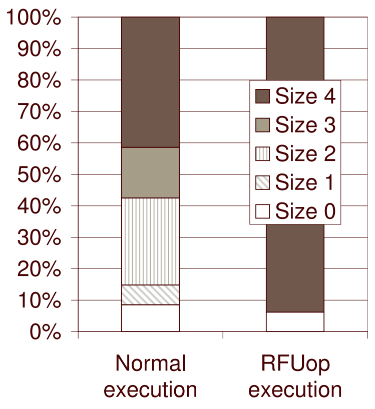
Fig. 3. *Performance comparison between a (4-issue, out-of-order) superscalar core with the same core augmented with an RFU*

by the optimizer. In our present compiler, this filter uses a simple heuristic that computes a quick estimate of the Cycles-per-instruction (CPI) of the RFU. If this CPI is greater than a threshold, then the RFU is rejected.

The compiler translates the selected RFUops to hardware implementations by aggressively speculating and exposing various parallelism forms, such as bit-, instruction-, pipeline- and loop-level parallelism [2]. Currently, it generates a C program that represents a custom timing and functional simulator of the RFUop; a Verilog back-end is in progress.

The compiler is implemented within the SUIF1 compiler infrastructure [13]. The SUIF-based compiler partitions the code and generates a lower level C program that is then compiled with gcc and simulated for measurements. Our experience with this tool flow indicates that the interaction between the base SUIF passes and gcc tends to introduce extraneous instructions in the application, creating a substantial amount of noise in our measurements. For instance, SUIF introduces a number of temporaries in the source code, which causes gcc to create register spills. For the decoder version of the `epic` benchmark from Mediabench [8], compiling with SUIF and then gcc causes the the execution time to increase by 10% compared to when it is compiled just with gcc. Further, the number of memory references increases by about 30%.

We use a modified version of the SimpleScalar simulation infrastructure [1] to measure the performance of the various benchmarks on our architectural model. Our simulation framework has two components—RFUop Simulation and CPU simulation. For the former, we use the automatically generated RFU simulator, as described above. A modified version of the SimpleScalar out-of-order simulator is used to perform cycle-accurate CPU simulation [12]. The ISA is augmented to include `inputrfu`, `startrfu` and `outputrfu` instructions. When a `startrfu` instruction is encountered, a separate RFUop simulation thread is invoked.



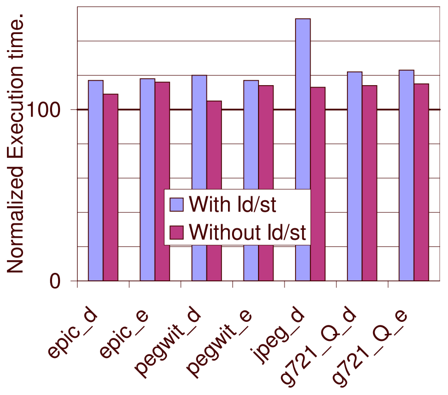


Fig. 4. *Relative breakdown of total execution time by Fetch Queue sizes. The fetch queue is almost always full during RFUop execution, implying that the pipeline mostly stalls during RFUop execution*

Fig. 5. *Effect of RFUops containing memory references.*

5 Experimental Results

We set up our experiments to compare the performance of our hybrid system with a CPU core. We compiled benchmarks from the Mediabench [8] and SPEC95 [11] benchmark suites. Most programs in the Mediabench suite have an encoder and a decoder version; these are indicated with `_e` and `_d` suffixes respectively. In the graphs where we compare performance, we have always normalized the execution times.

Figure 2 compares a hybrid system with a single-issue, in-order core, and the same core augmented with an RFU. We can see that in almost all cases the hybrid system outperforms the CPU-only version. Figure 3 compares a 4-issue, out-of-order superscalar processor, and an RFU augmented to the same processor. We compare two kinds of RFUop selection policies in these figures—when only loops are chosen, and when custom instructions are selected. In spite of the fact that the same compilation and mapping techniques were used in the two hybrid systems, we see that the performance of the superscalar+RFU system does not always match the performance of the superscalar processor. This implies that the techniques employed in the in-order hybrid architecture do not scale to the superscalar hybrid architecture. We attribute the slowdown to four principal factors:

- Parallelism: Currently, we do not reschedule any independent instructions between the `inputrfu`/`startrfu` and `outputrfu` instructions. Hence, the re-order buffer becomes a structural hazard when an RFUop is executing. Moreover, our current model does not support concurrent execution of multiple RFUops. This can severely affect the CPU pipeline. Figure 4 shows the relative breakdown of total execution time (of the adpcm_d benchmark) by fetch queue sizes. Notice that the fetch queue is almost always full during RFUop execution, which implies that the CPU is unable to issue any instructions as the pipeline stalls. Figure 6 shows the effect of stalling the CPU when an RFUop executes. In one case, the CPU is not allowed to issue any instructions after a `startrfu` is dispatched. The second is the default case when in-

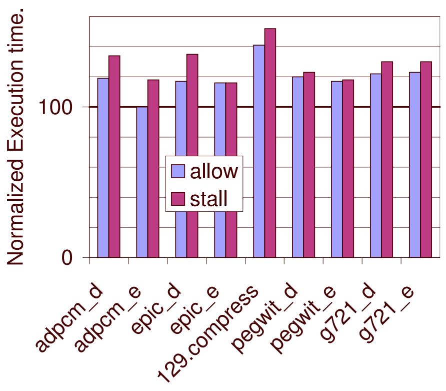


Fig. 6. *Effect of stalling CPU during RFUop execution*

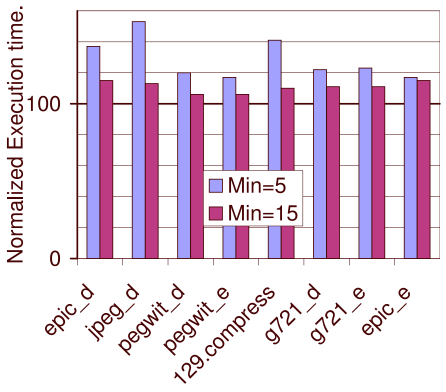


Fig. 7. *Effect of restricting the minimum RFUop size (in terms of number of instructions)*

structions are issued, but not committed until the `outputrfu` commits. Even with the restricted model in the latter case, there is up to 18% performance speedup. This shows that there is a lot of opportunity to issue more instructions in the CPU during RFUop execution. If the compiler can reorder instructions such that independent instructions are inserted between `startrfu` and `outputrfu`, the performance can only improve.

- RFUop selection: The code partitioning pass needs to be refined. The policy filter may need to be different. Selecting just instruction sequences or loops restricts us to a smaller domain of candidate RFUops. Moreover, our heuristic for diagnosing RFUops is simplistic. This tends to generate RFUops that produce more overhead than benefits. In Figure 7, we describe the effect of restricting the minimum RFUop size. When the minimum size of all RFUops is 15 static instructions, the performance is much better than when the minimum RFUop size is 5. Using a simple policy decision, we see that we can reduce some of the unnecessary overhead.
- Speculation: If an RFUop contains an entire loop, the compiler synthesizes a circuit that supports extensive pipelining and speculation. However, we do not speculate across loop iterations. This can cause a cascaded effect on performance if the critical path of the loop body contains memory references (with potentially large latencies). On the superscalar processor, speculation is supported across all conditional branches. Figure 5 shows that the performance of the system is better when RFUops don't contain memory references.
- Our tool flow introduces a substantial amount of noise. Hence, in some cases, the results are not indicative of a realistic evaluation.

6 Related Work

We can classify previous research based on the host CPU: those that augment a in-order core, and those that use a superscalar processor.

PRISC [9] is the first to explore the feasibility of a RFU on the datapath of a MIPS processor. All custom instructions must fit within a two-input/one-output signature, read and write the register file, and take a single processor cycle to execute.

GARP [3] augments an RFU co-processor to a MIPS core. The compiler attempts to accelerate loops by pipelining them using techniques similar to those used in VLIW compilers. The RFUops are chosen from hyperblocks that have loops embedded within them. The RFU can access the processor's caches and main memory directly.

The T1000 architecture [14] is an extension of the PRISC, although the CPU is a superscalar processor. The focus of this work is on developing a selective algorithm that is used to extract RFUops. Each RFUop is always assumed to execute in a single cycle, no matter how big it is, and the RFUops are not allowed to contain memory instructions. Also, the architecture has multiple RFUs on its datapath. Hence, concurrent RFUop execution comes naturally.

Chimaera [6] uses a superscalar processor in its model. It relies on fine-grained optimizations to obtain speedups. Hence, the compiler tries to extract sub-word parallelism from ALU instructions and map such sequences to RFUops. Memory instructions are not included in the RFUops.

OneChip [5] focuses on exploiting ILP in streaming multimedia applications. The host CPU is a superscalar processor. The architecture supports DMA, and assumes that all RFUops access a regular memory block, perform certain operations, and write back results to another regular memory block. Their compiler doesn't extract candidate RFUops, but requires the programmer to use a library, which contains calls to load configurations and access memory. These calls make all RFU memory accesses explicit, regular and predictable. Memory coherence is maintained by using techniques similar to those used in multiprocessors to maintain cache coherence.

7 Conclusions

While the idea of integrating a reconfigurable functional unit (RFU) into a general-purpose RISC processor has been studied extensively in the past, there have been few studies that have examined the new issues that arise when augmenting a high-performance superscalar architecture with a reconfigurable fabric. In fact, previous spectacular results often overlook progress that has been made in traditional architectures. Many previous studies either hand-code applications, ignore the extra ILP that can be extracted by out-of-order processors, or ignore negative interactions between an RFU and the main pipeline of the CPU.

Our work focuses on how to positively integrate an RFU into a superscalar pipeline. We target general-purpose applications and require automatic compilation of programs to the target hybrid architecture. The critical component of such a system is the compiler that can automatically map applications written in high-level languages to such architectures. The compiler must be cognizant of the factors that could influence performance. It is imperative that the compiler employs good heuristics in the code partitioning phase, so that performance is maximized and overhead is minimized. When using a shared memory space, memory consistency is a factor. While consistency must be maintained, the compiler must attempt to minimize the negative impact this can have on performance. In superscalar architectures, it is important to exploit all system resources and to minimize pipeline stalls. It is clear that much work needs to be done before an RFU can successfully boost total performance on an entire application.

Acknowledgments

This research is funded by the National Science Foundation (NSF) under Grant No. CCR-9876248.

References

1. Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, February 2002.
2. Mihai Budiu and Seth Copen Goldstein. Pegasus: An efficient intermediate representation. Technical Report CMU-CS-02-107, Carnegie Mellon University, May 2002.
3. Timothy J. Callahan and John Wawrzynek. Instruction Level Parallelism for Reconfigurable Computing. In Hartenstein and Keevallik, editors, *FPL'98, Field-Programmable Logic and Applications, 8th International Workshop, Tallinin, Estonia*, volume 1482 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1998.
4. André DeHon. The density advantage of configurable computing. *Computer*, 33(4):41–49, April 2000.
5. Jorge E. Carrillo E. and Paul Chow. The effect of reconfigurable units in superscalar processors. In ACM/SIGDA, editor, *Ninth ACM International Symposium on Field-Programmable Gate Arrays (FPGA'01)*, pages 141–150, February 2001.
6. S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera reconfigurable functional unit. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '97)*, pages 87–96, April 1997.
7. Michael Hind and Anthony Pioli. Evaluating the effectiveness of pointer alias analyses. *Science of Computer Programming*, 39(1):31–55, January 2001.
8. Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Micro-30, 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, 1997.
9. R. Razdan and M. D. Smith. A high-performance microarchitecture withv hardware-programmable functional units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–80. IEEE/ACM, November 1994.
10. Hartej Singh, Guangming Lu, Eliseu M. C. Filho, Rafael Maestre, Ming-Hau Lee, Fadi J. Kurdahi, and Nader Bagherzadeh. MorphoSys: case study of a reconfigurable computing system targeting multimedia applications. In *Proceedings of the 37th Conference on Design Automation (DAC-00)*, pages 573–578, NY, June 5–9 2000. ACM/IEEE.
11. Standard Performance Evaluation Corp. *SPEC CPU95 Benchmark Suite*, 1995.
12. Suraj Sudhir. Simulating processors with reconfigurable function units. Master's thesis, Carnegie Mellon University, Electrical and Computer Engineering Department, Carnegie Mellon University, Pa 15213, May 2002.
13. R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. In *ACM SIGPLAN Notices*, volume 29, pages 31–37, December 1994.
14. Xianfeng Zhou and Margaret Martonosi. Augmenting modern superscalar architectures with configurable extended instructions. In José Romlin et al., editor, *Proceedings of 15th International Parallel and Distributed Processing Symposium*, pages 941–950. Springer-Verlag, Berlin, May 2000.