A DAG-Based Design Approach for Reconfigurable VLIW Processors

Cesare Alippi William Fornaciari Laura Pozzi Mariagiovanna Sami
Dip. di Elettronica e Informazione, Politecnico di Milano
Milano, ITALY 20133
{alippi,fornacia,lpozzi,sami}@elet.polimi.it

Abstract

Embedded system design is a complex activity requiring to tradeoff a number of parameters such as cost, performance, flexibility, power and time-to-market. This paper explores the possibility of enabling a partial customisability of the Instruction Set of VLIW processors for embedded applications, by exploiting FPGA technology. In particular it is presented a formal methodology to select the application critical parts, whose RFUs (Reconfigurable Functional Units) implementation allows to reduce the overall execution time. Experiments conducted on representative benchmarks show the viability of the proposed approach.

1 Introduction

The design of an embedded system is a multi-goal activity with a search space constrained by not purely functional issues. To quickly follow the market needs, the typical tradeoff is not only related to cost and performance, but also includes an evaluation of the flexibility in terms of architecture and design methodology. One of the most important decisions concerns the choice between the programmability of a general component as the microprocessor and the performance of dedicated hardware (e.g. an Application Specific Integrated Circuit). The former solution supports a broad range of possible applications, while the latter allows to easily satisfy application constraints such as performance, size and power consumption. To cope with such problems, the hw/sw codesign suggests to concurrently balance, at system-level design time, the presence of hw and sw in the final application. Other approaches address the use of microprocessor with partially customisable IS (Instruction Set), known as ASIP [11], to tune the microprocessor characteristics towards the real operating requirements of the specific application. Further flexibility has been made available by current FPGA technology, which supports run-time reconfiguration of hardware with complexity and performance sufficient to operate around 100MHz/200Kgates per subsystem [10, 2]. There are, however, open issues to be addressed such as poor cost-effectiveness of FPGAs for volume production with respect to other technologies. Another open issue is the compilation strategy to be adopted when reprogrammable CPU architectures are envisioned, especially when they exhibit pipelining or Instruction Level Parallelism exploitation. In fact, retargetable compilers are not yet a mature discipline (in particular in terms of marketable CAD tools) and the achievement of acceptable results still requires strong assumptions on the class and flexibility of the considered architectures [5]. The approach we suggest is a *meeting in the middle*, aiming at increasing the freedom in exploring the design space while optimising and simplifying the compiler tasks. Since present technology allows integration of conventional CMOS structures and FPGA sections on the same chip, we envision a processor with multiple Functional Units (FUs), some of which will be application-specific Reconfigurable Functional Units (RFUs), implemented through FPGA technology. We consider the presence of a partially customisable *core* CPU including a certain set of Functional Units, capable of a native IS that is customised (extended) via the *reprogrammable* FUs. In this way, we overcome the complexity of designing a fully application-specific CPU, involving not only the hw design effort but also the development of tools such as compilers, simulators, etc, enabling, in turn, the use of customisable cores also for cost-sensitive designs. Our target applications (e.g. image processing, cryptography) require not only volume production, but also good design flexibility during their lifetime. Performances are critical and there is a certain degree of *medium term* predictability (e.g., in the range of a single coarse grain cluster of data) of the operations to be computed.

The achievement of reconfigurable processing by using programmable hw is not new; some promising investigations have been carried out during last decade. In [8], an RFU working in parallel with the normal units is mapped onto an on-chip FPGA implementing different multi-operands functions. The program includes calls to the RFU that, should the instruction not yet be available on the unit, stall the microprocessor until reconfiguration is completed. An approach working at board-level is discussed in [14], where a processor and an off-the-shelf partially reconfigurable FPGA is used to minimise the time necessary to reprogram the device when a given instruction in not mapped. The system described in [9] consists of a workstation coupled with a coprocessor consisting of a VLIW processor with three FPGA units. All dependency checks and instruction level parallelism identification is left to the programmer. An aspect of the work described in [13] is similar to ours, in that the problem of automatically finding the function to be mapped onto RFUs is addressed. However, such function is constrained to single cycle execution, while our approach overcomes this constraint. The novelty of our solution with respect to the above proposals is that it relates to a more general methodology, rather than a specific application. We identify the analysis framework to set-up a design flow able to select the best candidate functions to be implemented via the RFUs.

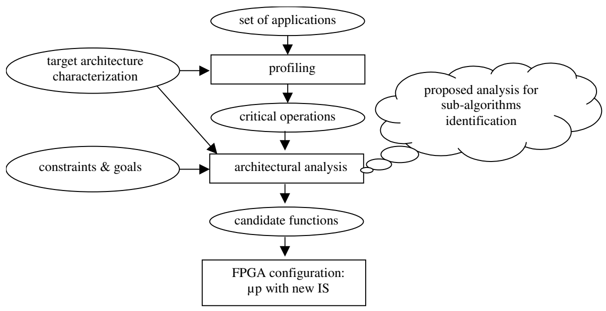

Figure 1. The proposed design flow

A skeleton of the proposed design flow is depicted in Figure 1. At the topmost level there is a profiling activity, which identifies the distribution of the computational workload among the different functions composing the application. Results of this phase provide a set of guidelines for the following architectural analysis that will identify the FPGA configuration, i.e. the set of functions to be implemented onto the RFUs, according to a speedup/cost tradeoff. During architectural analysis the following aspects are evaluated:

- granularity and implementability of the candidate functions, according to the chosen architecture of the RFUs (see section 3);
- expected speedup, balanced with the reconfiguration overhead;
- analysis of the cross-relations between the compilation strategy for the VLIW core and the modified IS (new *opcodes* are now available for the compiler);
- identification and optimisation of the configuration scope, i.e. the part of the application where it is used the same RFU configuration.

The focus of this paper is on the most critical part of the proposed design flow, i.e. the construction of a theoretical model and of a strategy to identify the candidate solution to be implemented via RFUs. In particular, Section 2 describes the target VLIW architecture we are extending via use of RFUs while section 3 details the adopted DAG (Direct Acyclic Graph) representation of the algorithm to be analysed and the properties of the substructures we identified (called MISO) to select the candidate functions to be mapped onto the RFUs. Some experimental results showing the viability of the proposed methodology are reported in Section 4. Finally, Section 5 is devoted to conclusions and future directions.

2 The Target Architecture

Our target architecture is a VLIW processor augmented with one (or possibly more) RFUs. A VLIW machine [12] is capable of issuing and executing multiple operations per cycle, bundled in a "MultiOp" long-word instruction, and it relies on compile-time scheduling to determine independent operations which can be issued concurrently at execution time. The machine is equipped with multiple functional units so as to exploit the parallelism that has been exposed by the compiler. Many techniques for exploiting instruction-level parallelism have been studied and implemented in optimising compilers [1], in order to find groups of independent operations in each cycle.

Because of the existence of a high number of FUs, advanced VLIW processors are organised in clusters. Every cluster consists of a number of functional units and a register file that these FUs share. Since the cost of register files grows exponentially with the number of read/write ports, the organisation in clusters solves the cost problem by keeping the number of ports per register file low. The IS architecture provides special instructions that copy values between register files of different clusters [3].

The RFU, being implemented by means of FPGA technology, is reprogrammable. It represents an additional datapath of the processor and can execute a specialised instruction that is generated 'ad hoc' from the application, through the methodology discussed in section 3. It is reasonable to assume that the RFU represents a new cluster of the VLIW architecture, and therefore has a dedicated register file that shares values with other clusters through *copy* instructions. It has internal memory which is preloaded in the reconfiguration phase. This can be used for holding inside the FPGA values that are generated and consumed only by the RFU and not shared with other functional units: as such, they do not need to be written in the register files. This releases the processor load/store pipes and busses from unnecessary traffic.

The specialised operation executed by the RFU implements a segment of computation extracted from the application algorithm and mapped onto the FPGA, as outlined by the design flow in Figure 1. A new opcode, the *fpga-opcode*, is correspondingly generated and it replaces the relevant segment of computation in the translation from high level code into machine code. The new *fpga-opcode* is made available to the compiler as an extension of the machine IS, and information such as latency (which must be known for scheduling) and number of operands required is also given. Reconfigurations of the FPGA are driven by the compiler prior to execution of the application code, or possibly at the beginning of every section of code that requires reconfiguration (studies of a policy of configuration prefetching have been published [7]). This policy allows to avoid exceptions occurring because of a *configuration miss* (which is the case considered in [8]): every occurrence of an *fpga-opcode*, in a code segment which lies between two consequent reconfigurations, always refers to the specialised operation which is currently mapped onto the FPGA.

One remark must be made about the format of the FPGA operations. An obvious requirement is that these operations will 'fit' in the multi-op instruction format; this constrains, in particular, bitlength of the FPGA operations. On the other hand, FPGA operations may easily involve a larger number of input operands than the IS elements; such number may be limited only by feasibility constraints of the register file, suggesting at present an upper bound of eight input ports to keep reasonable efficiency. Methods for accessing FPGA operands could be the use of 'implicit' (i.e. predeterminated) registers as source registers, or the management of the FPGA register file as a stack.

Assuming, then, this target architecture (with all relevant constraints and requirements as detailed in Section 1) our problem becomes that of extracting from the application algorithm the segments of computation that are best candidates to be implemented as *fpga-opcodes*. The approach we propose, described in the next section, identifies such best candidates based on the graph (more precisely, the DAG) corresponding to the application, from which suitable candidates subgraphs are extracted.

3 DAG Analysis and MISOs

Before describing the approach proposed for identification of the candidate functions to be mapped onto RFUs, we discuss some issues constraining the design of such RFUs. Basically, these restrictions aim at simplifying synchronisation between RFU and core CPU without adding undue burdens, particularly to the control unit.

Thus:

1. RFUs never directly access the data cache (this avoids, in particular, exceptions on loads caused by RFUs);
2. An RFU is provided with a 'visible' register file (RF), accessible to the core CPU by means of copy instructions; it may also have a 'private' internal memory storing data that are accessed only by the RFU and that are typically written in the memory itself at configuration time (these are seen as 'local variables')
3. Multiple read operations but a single write access on the visible RF are allowed at any given time to simplify register file structure and interaction with the other functional units.
4. Operations performed by the RFU cannot be interrupted; in the case of an exception raised outside the RFU, the RFU is simply stalled without modifying its computational status.

Identification of candidate functions to be mapped onto the RFU is performed by analysing the DAG of the application algorithm or, more properly, the DAGs of the *critical basic blocks* identified in the profiling phase outlined in Section 1 (from now on, by DAG we will refer to such set of 'critical' sub-DAGs). By considering basic blocks, we assume a linear control flow[1]. Our aim is not so much to define *complex instructions* as to identify *sub-algorithms* critical for application performances and that can be usefully mapped onto a dedicated RFU. Such sub-algorithms appear as subgraphs of the given DAG. Given the characteristics of the DAG, nodes have two inputs at most (they represent assembler-like operations) and fanout greater than one is allowed, i.e. the output produced by a node can be input to multiple destination nodes.

Restrictions 1 and 3 above lead us to consider only *connected* subgraphs that have a priori unlimited number of inputs and a single output. We call these subgraphs MISO (Multiple Input Single Output). Considering non-connected subgraphs for a single RFU would not be a suitable choice, since such subgraphs could well exploit the parallelism among multiple functional units.

A formal definition of a MISO M^i is as follows. Denote by $G^i = < V^i, E^i >$ a subgraph where V^i is the set of nodes in G^i and E^i is the set of all edges departing from such nodes. An edge $e^i \in E^i$ is identified by its source node ($v^i_k \in V^i$) and its destination node v^i_l and it is denoted by $e^i(k,l)$. If for all $v^i_k \in V^i$ excepting one node v^i_O it is true that:

$\forall e^i(k,l) \in E^i, v^i_l \in V^i$

then G is MISO. We denote the MISO as M^i, v^i_O being its *output node*. Note that any node in V^i may have incoming edges originating from nodes not belonging to V^i.

[1]The solution can actually be extended to blocks including data-independent loops.

As expected and experimentally shown in Section 4, the speedup obtained by implementing a MISO on FPGA is proportional to its size. Thus we are interested in a MAXMISO MM^i, defined as a MISO that is not completely included in any other MISO. It is proved that:

Theorem 1 *Two MAXMISOs MM^i, MM^j cannot partially overlap.*

Proof:

The theorem is proved by contradiction. Assume $MM^i \cap MM^j \neq \Phi$. This implies that at least one node v^* in the DAG belongs to both MM^i and MM^j. Then the alternative possibilities are:

1. v^* is the output node for either MM^i or MM^j or both. Then, given the definition of MISO, there is a MISO $MM^k = MM^i \cup MM^j$. As a consequence, MM^i and MM^j are not maximum, contrary to the assumption.
2. v^* is not an output node for either MM^i or MM^j. Then, v^* could not be the source of edges connecting to nodes in MM^i (otherwise MM^j would have more that one output and therefore would not be a MISO) nor to nodes in MM^j (MM^i would not be a MISO); as a consequence, v^* could only belong to a subgraph not connected within both MISOs, contrary to initial assumption.

■

The above property can be used as the basis for an algorithm (later described) that extracts MAXMISOs from all critical blocks in a DAG.

Some operations can be performed on MAXMISOs thus built. In particular, variables that appear as inputs only within a MAXMISO (read from memory, not resulting from previous processing steps) can be considered as *local* variables (typically weights used in DSP algorithms fall within this category); the number of MISO inputs can be correspondingly decreased. All MAXMISOs thus modified are then listed and examined:

1. A number of iterations f^i is associated with each MM^i (it corresponds to the number of iterations of the related critical block, statistically evaluated in the profiling phase).
2. Isomorphic MAXMISOs are tagged.
3. If a MAXMISO MM^i is isomorphic to a MISO $m^k \subset MM^j$ ($j \neq i$), MM^i and MM^j are suitably tagged (obviously MM^i and MM^j do not share any node; they can either have been extracted from different critical blocks or be disjoint subgraphs from the same critical block).

At this point, each MAXMISO is checked for *feasibility* , in terms of number of input variables and complexity of resulting structure (roughly evaluated by a high level synthesis phase applied to the MAXMISO itself, thus providing a preliminary area and latency estimation).

The upper bound on the number of inputs depends on different implementation alternatives that can be adopted for the RFU, such as multi-phase access to the register file rather than single-phase access.

Consider a non-feasible MAXMISO MM^j; if it has been tagged in step 3 above (i.e. it contains at least one subgraph m^k isomorphic to a separate MAXMISO MM^i), the relevant subgraph(s) m^k is considered for RFU mapping if feasible (of course, it inherits the number of iterations associated with MM^j); otherwise a search for feasible MISOs is performed, *restricted to unfeasible MAXMISOs*; in fact:

Lemma 1 *All MISOs in the DAG are either MAXMISOs or contained in a MAXMISO.*

The proof is immediate.

In the following the algorithm for the identification of all MAXMISOs in a DAG is presented. The algorithm, which shows linear complexity, has been implemented on top of the SUIF environment [6].

```
∀Node ∈ Nodes_to_be_analysed do
{
  Generate_MAXMISO(Node)
  Nodes_to_be_analysed −= Nodes_in_MAXMISO
}

Generate_MAXMISO(Node)
{
  ∀Parent_of_Node do
  {
    if (fanout == 1) {
      include(Parent_of_Node)
      Generate_MAXMISO(Parent_of_Node)
    }else
      fanout −−
  }
}
```

Figure 2. Pseudocode of the algorithm for the extraction of all MAXMISOs in a DAG

Figure 2 shows the pseudocode for the generation of all MAXMISOs within the DAG. The algorithm operates in two steps: first, a node is chosen to be the *exit_node*, then the program activates a function which builds the MAXMISO related to such *exit_node*. Exit nodes are chosen *upwards*, i.e. starting from the exits of the DAG. Initially, the set of *Nodes_to_be_analysed* coincides with the set of nodes of the DAG; afterwards, when a MAXMISO has been generated, its nodes are removed from the *Nodes_to_be_analysed* set. The function `Generate_MAXMISO` starts from the chosen *exit_node* and recursively tries to include its parents in the MAXMISO being generated. Recursion ends when the encountered node is non legal (e.g. it is a load instruction) or has a non reconvergent fan-out.

In order to maximise code efficiency, the latter case has been implemented as follows. Every time a node with fanout greater than one is visited by the function, its value *fanout* is decremented by one. Now, if and only if the node is visited *fanout* times, (and therefore one function call visits the node when its *fanout* value has been decremented down to one) the node is included in the MAXMISO (since this implies reconvergence). If the node is visited a number of time smaller than *fanout*, then it can only be an exit node for a different MAXMISO, and it is not to be included in the current one. The proposed algorithm shows a complexity linear with the number of nodes in the examined DAG. This follows from Theorem 1: since MAXMISOs cannot overlap, each node is visited only once.

In the case several feasible MAXMISOs are available within the basic block and the FPGA does not allow physical implementation for all of them, a subset of them must be selected. As a figure of merit to compare candidates MM^i we consider:

$max_i[(T^i_{FUs} - T^i_{RFU}) \times occ_i]$

where T^i_{FU} and T^i_{RFU} are the time required to execute a single MM^i on the basic FUs and on the RFU respectively, and occ_i is the number of occurrencies of MM^i in the examined DAG. Note that the final speedup depends on the resulting scheduling of the whole application, while an evaluation of many candidates can only be made locally, i.e. without regard to the overall scheduling.

4 Experimental Results

Extraction of MAXMISOs by the described algorithm has been performed on a number of significant applications in order to carry out a preliminary evaluation on dimensions and morphology of the resulting MAXMISOs. The C code of the applications has first been compiled by SUIF in order to generate an intermediate representation of the program where each instruction is broken down into assembler-like operations. The next step was to extract each critical basic blocks by means of a profiler. The DAGs of such critical blocks were then processed by the algorithm that identifies the contained MAXMISOs. Table 1 summarises some features of the four main MAXMISOs found in the DSP application given in [4] and a DES cryptography algorithm.

We characterise each MM with its number of nodes (#N), number of inputs (#I) and number of distinct occurrences in the DAG (# occ).

It is interesting to note the significant occurrence of MAXMISOs in some relevant applications, such as cryptography (up to 32 times in the same critical block). A second interesting feature is that the number of inputs, a priori unlimited, is always reasonably small; this property makes the MMs good candidates for RFU implementation. Finally, it can be noted that some relevant MAXMISOs are included in others, as detailed in the last column of Table 1; when this happens the number of occurrences of the included MAXMISO is augmented by the number of occurrences of the including one.

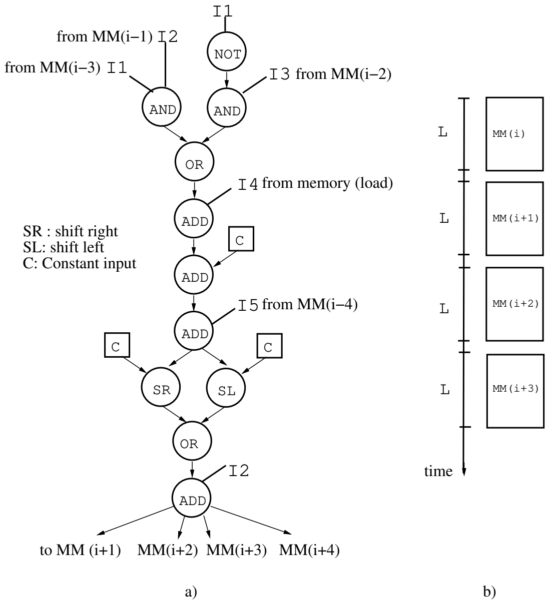

Figure 3. a) MAXMISO C for the cryptography algorithm and interactions with other occurrences of the same MAXMISO (these are referred to as MM(i) ($0 <= i <= 32$)). b) Resulting scheduling.

To provide further details on design considerations and speedup based on extracted MAXMISOs, let us refer to the cryptography case, whose MAXMISO C is shown in Figure 3a. A critical path from input I2 (top left) to the exit is identified and the corresponding scheduling is shown in Figure 3b. Note that, in this case, the FPGA unit cannot be pipelined because each occurrence of a MAXMISO depends on the output of the previous one. In order to calculate the speedup obtained by MAXMISO implementation on RFU, let us compare the latency of the resulting RFU with the number of cycles required to execute, in the basic FUs, the segment of computation corresponding to the MAXMISO.

We can calculate the RFU latency (denoted by L in Figure 3a) as the sum of one control step for each adder, and one for each group of logic operations, adding up to 6 control steps. In the case of execution on the basic FUs, 8 cycles are required (one for each operation in the critical path); this time increases to 16 cycles if forwarding is not implemented. If we denote by τ_c the clock cycle of the processor and by τ_s the control step duration for the FPGA, the gain obtained by implementing MAXMISO C in the RFU is equal to $(16\tau_c - 6\tau_s)$. For a reasonable and maybe pessimistic value τ_s/τ_c of

	MAXMISO A			MAXMISO B			MAXMISO C			MAXMISO D			
application	#N	#I	#occ	#N	#I	#occ	#N	#I	#occ	#N	#I	#occ	Isomorph.
TriCubic	9	5	3	10	5	3	11	6	3	12	6	3	$A \in B, C \in D$
Quarcube	6	7	1	8	9	1	3	4	1	5	5	1	$A \in B, C \in D$
len4	20	6	1										none
InvSqrt	7	4	1	16	6	1	3	1	1	6	2	1	$A \in B, C \in D$
Mul32	7	4	1	8	4	1							$A \in B$
Criptography	7	6	16	12	7	16	11	5	32				$A \in B$

Table 1. Characteristics of MAXMISOs extracted form various application programs

2, the gain becomes equal to 4 processor cycles. This must be multiplied by the number of occurrences of the MAXMISO in the DAG (32 in this case) and for every iteration of the envisioned critical block. Note that the processor FUs are now available for execution of possible parallel threads; this is a source of additional speedup.

Last it can be noted that the RFU implementation of a segment of computation can benefit, when compared to execution in presence of the standard FUs only, from the following potential optimisations:

1. Chaining: when two or more simple operations such as shifts or bitwise logic are in cascade, they can be chained together so that they can execute in a single clock cycle.
2. Constant collapsing: when the input of a MISO is a constant, this can be held in the FPGA memory so that it does not 'occupy' a register file read port. In the case the constant is always the same value in every occurrence of isomorphic MISOs, it can collapse in the implementation so that resulting area and delay are lower.
3. Availability of intermediate results: they do not need to be written back and reread, and a cycle is saved at every stage with respect to those architectures where forwarding is not implemented.

5 Conclusions

Augmenting a processor with one or more Reconfigurable (and therefore customisable on the fly) Functional Units results in the possibility to better tailor resources *to each* different application served. When such architecture is envisioned, an important point is to automatically identify the function that is to be implemented in the RFUs. An algorithm for extraction, from a DAG, of segments of code to be executed on the RFU has been presented in this paper and has been applied to DSP based applications. Such segments of code are of the form of MAXMISOs, a maximum graph with Multiple Inputs and Single Output. MAXMISOs can be extracted from the DAG with linear complexity. Further research efforts will be addressed to the search of 'kernel' MISOs which are common elements to many MAXMISOs, a case that we have experimentally noticed to often occur.

References

[1] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler Transformations for High Performance Computing. *ACM Computing Surveys*, pages 345–420, 1994.

[2] D. Bursky. High-Density FPGA Family Delivers Megabyte Capacity. *Electronic Design*, 45(25):67–70, November 1997.

[3] G. Desoli. Instruction assignment for clustered VLIW DSP compilers: a new approach. *Hewlett Packard Technical Report, HPL-98-13*, 1998.

[4] Andrew Glassner. *Graphic Gems I-V*. Graphic Gems book series. Academic Press, 1992.

[5] G. Goossens, J. Van Praet, D. Lanneer, W. Geurts, and F. Thoen. Programmable Chips in Consumer Electronics and Telecommunications. *NATO ASI on Hw/Sw Codesign*, pages 135–164, 1995.

[6] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, December 1996.

[7] S Hauck. Configuration Prefetch for Single Context Reconfigurable Coprocessors. *ACM/SIGDA International Symposium on FPGAs*, 1998.

[8] S. Hauck, T.W. Fry, M.M. Hosler, and J.P. Kao. The Chimera Reconfigurable Functional Unit. *IEEE Proc. FCCM*, 1997.

[9] C. Iseli. *Spyder: a Reconfigurable Processor Development System.* PhD thesis, Ecole Polytechnique Federale de Lausanne, 1996.

[10] D. Lewis, D. Galloway, M. van Ierssel, J. Rose, and P. Chow. The Transmogrifier-2: a 1 Million Gate Rapid Prototyping System. *ACM Proc. Symposium on FPGA*, pages 53–61, February 1997.

[11] C. Liem, T. May, and P. Paulin. Instruction-Set Matching and Selection for DSP and ASIP Code Generation. *Proc. IEEE EDAC*, pages 31–37, 1994.

[12] B. Rau and J. Fisher. Instruction-Level Parallel Processing: History, Overview and Perspective. *The Journal of Supercomputing*, 7:9–50, 1993.

[13] R Razdan and M. D. Smith. A High-Performance Microarchitecture with Hardware-Programmable Functional Units. *Proc. MICRO-27*, pages 172–180, 1994.

[14] M.J. Wirthlin and B.L. Hutchings. A Dynamic Instruction Set Computer. *IEEE Proc. FCCM*, pages 99–107, 1995.