Instruction Set Extension with Shadow Registers for Configurable Processors

Jason Cong, Yiping Fan, Guoling Han, Ashok Jagannathan, Glenn Reinman, Zhiru Zhang

Computer Science Department, University of California, Los Angeles

Los Angeles, CA 90095, USA

{cong, fanyp, leohgl, ashokj, reinman, zhiruz}@cs.ucla.edu

ABSTRACT

Configurable processors, which allow customization and extension of the base instruction set architecture for a specific application or a domain of applications, are becoming increasingly popular for modern embedded systems (especially for the field-programmable system-on-a-chip). While steady progress has been made in the tools and methodologies of automatic instruction set extension for configurable processors, the limited data bandwidth available in the core processor (e.g., the number of simultaneous accesses to the register file) becomes a potential performance bottleneck.

In this paper we first present a quantitative analysis of the data bandwidth limitation in configurable processors, and then propose a novel low-cost architectural extension and associated compilation techniques to address the problem. Specifically, shadow registers are introduced to selectively copy the execution results in the write-back stage, which can efficiently reduce the communication overhead due to the data transfers between the core processor and the custom logic. To take full advantage of the extension, an effective shadow-register binding algorithm is presented to minimize the communication overhead. The application of our approach results in a promising performance improvement.

1. INTRODUCTION

Designing a modern embedded system in nanometer technologies is more difficult than ever. Due to the complexity and electrical design challenges posed by each new technology generation, the design productivity gap continues growing despite increasingly expensive CAD tools. This urges a move toward the use of programmable and configurable solutions to achieve a fast turn-around time and to accommodate various applications.

Reconfigurable platforms, combined reconfigurable fabric with a general-purpose processor, are a promising approach to combining the flexibility offered by a general-purpose processor and the speedup (and power savings) offered by an application-specific hardware accelerator. Generally, there are two ways to couple the reconfigurable fabric with the microprocessor [5]. Loosely coupled, a reconfigurable fabric can be used as a co-processor [21][11]. Co-processors perform more complicated tasks independently without the constant supervision of the main processor. The main processor sends the necessary data to the co-processor at the initialization stage. With the internal state registers, the co-processor does not need to transfer data during the computation period. On the contrary, application-specific instruction-set processors (ASIPs) tightly integrate the reconfigurable fabric as additional application-specific function units, thus extending the basic instruction set with the custom instructions. These augmented function units are used to exploit the instruction level parallelism within the specific applications, and the execution is still on the main processor's datapath. These hardware resources can be either runtime reconfigurable functional units [22] or pre-synthesized circuits [24]. Normally, they need direct access to or data transfer from the central register file in the main processor. The recent emergence of many commercially available embedded processors with both configurability and extensibility (e.g., Altera Nios/NiosII [23], Tensilica Xtensa V/LX [24], Xilinx MicroBlaze [25], etc.) testifies to the benefit of this approach. As an example, Figure 1 (taken from Altera's website [23]) shows the instruction logic of Altera NiosII [23]. This processor contains a RISC core as the base architecture, and the custom logic can extend the functionality of the ALU by implementing the custom instructions for complex processing tasks as either single-cycle (combinatorial) or multi-cycle (sequential) operations. In this paper we will focus on data bandwidth problem for ASIPs.

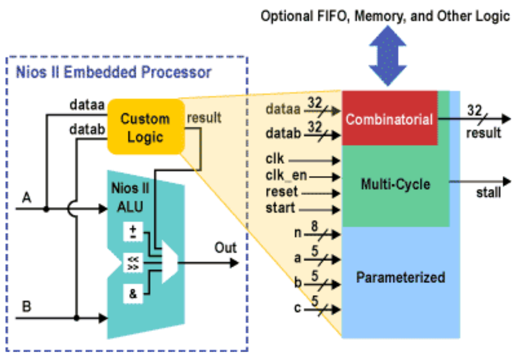


Figure 1. Custom instruction logic of NiosII.

The research community has also spent a considerable amount of effort in the ASIP area for almost a decade. A broad overview of

FPGA'05, February 20–22, 2005, Monterey, California, USA.
Copyright 2005 ACM 1-59593-029-9/05/0002...$5.00.

the ASIP design, its advantages, applications, and fundamental challenges can be found in [14]. A high-performance ASIP architecture is described in [22]. It integrates a fast reconfigurable functional unit into the pipeline of a superscalar processor to implement the application-specific operations. The ASIP architecture and compiler co-exploration problem is addressed in [8].

A crucial step to achieving high performance in an ASIP design is to select an optimal custom instruction set. However, for large programs, this is a difficult task to be managed by manual designs, and is further complicated by various micro-architectural constraints, such as the clock period, available chip area, etc. These constraints have motivated a large body of recent research to address the automatic instruction set extension problem.

A template generation, matching, and covering algorithm is proposed in [13] to automatically identify the custom instructions. The candidate templates are first generated by a clustering algorithm based on occurrence frequency. Then the directed acyclic graph covering is formulated as the maximum independent set problem to maximize the number of covered nodes using a minimum number of templates. Unfortunately, this work does not consider the architecture constraints during the template generation.

An approach presented in [19] generates and selects the candidate custom instructions from operation patterns in the data flow graph. The method was later extended in [20] to handle the complex control flows in the embedded software programs. A comprehensive priority function is computed to rank and prune the candidate instructions.

In [2] the candidate extended instruction is defined to be a convex directed acyclic subgraph subject to certain input and output constraints. A branch and bound algorithm is used to decide whether or not to include a node of the control data flow graph (CDFG) when creating the candidate. The time complexity of this approach grows exponentially as the problem size increases.

The extended instruction set synthesis technique proposed by [6] solves three sub-problems under the micro-architectural constraints: pattern generation which enumerates all the candidate instructions, pattern selection which selects a subset of the candidates to form the extended instruction set, and application mapping which maps the CDFG onto the extended instruction set. Particularly, the application mapping problem is transformed into the minimum-area cell-library-based technology mapping problem in the logic synthesis domain, which can be solved exactly through binate covering. Applications of this approach to several small data-intensive DSP applications on the soft core processor Nios show 2.75X speedup on average with little resource overhead (2.54%).

It is important to mention that although the existing techniques are efficient in identifying the most promising clusters of operations to be implemented by the custom instructions, most of the performance speedup (about 60%) comes from the cluster with more than two input operands (according to the study in [12]). This exceeds the number of read ports available on the register file of a typical embedded RISC processor core. Strictly following the two-input single-output constraint generally leads to small clusters with limited speedup.

Generation of larger clusters with extra inputs is allowed in [19][20] by using the custom-defined state registers to store the additional operands. Unfortunately, at least one extra cycle is needed for each additional input to be loaded into a custom-defined state register. The communication overhead due to these data transfers between the core processor and the custom logic can significantly offset the gain from forming a large cluster.

Our contributions in this paper are threefold. First, we present a quantitative analysis of data bandwidth limitation. Second, we propose using the shadow register as a novel low-cost architectural extension to mitigate the bandwidth limitation in the configurable processor. Third, we formulate a new shadow register binding problem and present an efficient algorithm to solve the problem.

The remainder of the paper is structured as follows. We first present the quantitative study on the data bandwidth problem in Section 2. Our proposed architectural extension and associated compilation techniques are described in Sections 3 and 4, followed by conclusions in Section 5.

2. ANALYSIS OF BANDWIDTH LIMITATION

2.1 Motivation

The architectural model targeted in this paper is a classical single-issue pipelined RISC core processor with a two-read-port and one-write-port register file (This is similar to the Altera Nios/NiosII micro-processor). Under this processor model, a custom instruction follows the same instruction format and execution rules, which include: (1) The number of the operands and results of a custom instruction is pre-determined by the extensible architecture; (2) The custom instruction cannot execute until the input operands are all ready; (3) The custom instruction can read the core register file only during the decode/execute stage, and can commit the result only during the write-back stage. This extensible architecture simplifies the implementation since the base instruction set architecture can remain unchanged.

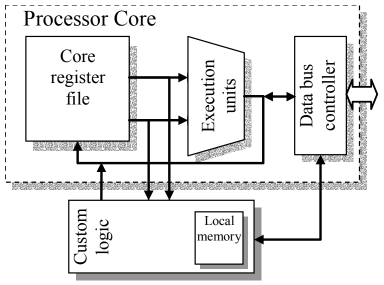


Figure 2. A typical extensible processor.

Figure 2 shows the block diagram of a typical configurable processor, where a two-operand instruction format is used. During the execution, the custom logic reads two operands from the core register file and writes the result back directly. This extensible architecture simplifies the implementation since the base instruction set architecture (ISA) can remain unchanged.

However, such a scheme would restrict the custom instruction to having only two input operands, thus limiting the complexity of the computations. Generally, when the input number constraint of the custom instruction is relaxed, more performance speedup can be obtained by clustering more operations in one custom instruction to exploit more parallelism. According to the study in [12], most of the performance speedup (about 60%) comes from the cluster with more than two input operands. Unfortunately, if a custom instruction needs extra operands, the processor core has to explicitly transfer the data from the register file to the local storage of the custom logic through the data bus. Only after that is the custom instruction allowed to execute. The communication through a data bus may take multiple CPU cycles, and significantly offset the performance gain by using the custom instruction. Therefore, the limited port number in the register file is a performance bottleneck of the extensible processors.

2.2 Evaluation Framework

In order to better understand the performance bottleneck in ASIP design, we developed an ASIP performance evaluation framework as shown in Figure 3. SimpleScalar [3], which is a cycle-accurate simulation tool set, is used to estimate the performance of the processor and the impact of communication cost. To have a quick evaluation of data bandwidth limitation, our ASIP compilation is applied on the compiled binary code of the benchmarks.

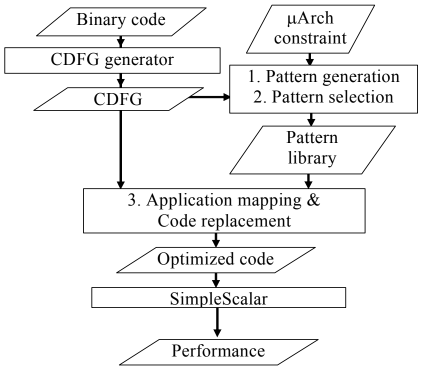


Figure 3. Our compilation and simulation flow.

Based on the execution trace generated by SimpleScalar, CDFG generator produces the control data flow graph. Under the given micro-architectural constraints, the ASIP compilation problem is solved in three steps based on [6]. The first step, called pattern generation, enumerates all candidate patterns from a given control data flow graph through the cut enumeration technique. Pattern selection is then performed in the second step. A cost function that considers the occurrence, speedup, and area is calculated to guide the selection. The selection problem is formulated as a 0-1 knapsack problem which is pseudo-polynomial time solvable via dynamic programming. In the third step, called application mapping, we map the data flow graph into the selected patterns to minimize the total latency. The application mapping problem is shown to be equivalent to the minimum area cell-library-based technology mapping problem in the logic synthesis domain, which can be solved exactly through binate covering. The algorithmic details of the aforementioned three steps can be found in [6]. With the optimized code as input, SimpleScalar simulates the program execution on the configurable processor and provides the performance estimation.

2.3 Analysis Results

In this study, we modeled a single issue, in-order RISC configurable processor which is similar to Altera Nios/NiosII [23]. Table 1 shows the detailed machine configuration. The instruction set allows two-input operands and one-output operand. The C examples used in the experiments are from Mediabench [15] and Mibench [10].

data cache L1	8KB, 4-way, 1-cycle latency
instruction cache L1	8KB, direct mapped, 1-cycle latency
unified L2 cache	256KB, 4-way, 8-cycle latency
register file	2 read ports, 1 write port
ALU	1-cycle latency
MULT	3-cycle latency
reconfigurable units	latency of the critical path of the collapsed instructions

Table 1. Detailed processor configuration.

As previously mentioned our ASIP compilation tool generates extended instructions and maps the program with the extended instruction set. All the extended instructions are generated within the basic block boundary. Memory operations are not allowed in any extended instruction. We assume that the latency of the extended instructions equals the latency of the critical path in the collapsed computation cone.

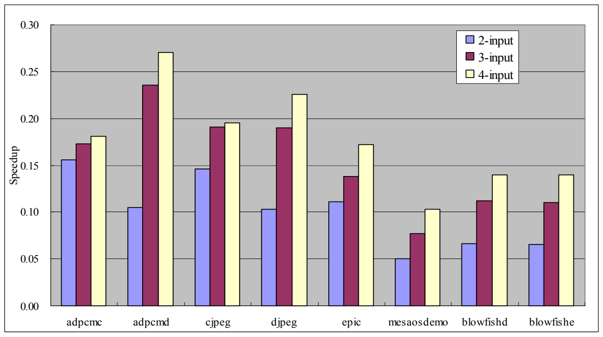


Figure 4. Ideal speedup under different input constraints.

Figure 4 shows the ideal speedup for the each benchmark under different input size constraints. The speedup is measured by comparing the number of simulated execution cycles of the program on the extended instruction set with the number of cycles of the original code on the basic instruction set. We assume that there is no limit on the number of read ports in the register file so that no move operations are needed.

The results shown in Figure 4 indicates that we can achieve 10% speedup on average with the 2-input constraint. Under 3-input and 4-input constraints, 15% and 18% speedup can be achieved, respectively. It also shows that for these examples, the designs under 3-input and 4-input constraints can achieve 50% and 80% more speedup over 2-input ones respectively.

However, the processor can only provide two simultaneous accesses from the register file. Move operations have to be inserted before the execution of 3-input or 4-input extended instructions. In our experiment, we assume that the move operation needs only one clock cycle. Figure 5 shows the speedup drop due to the move instructions, which is defined as

$$Speedup\_drop = \frac{Speedup_{ideal} - Speedup_{reg}}{Speedup_{ideal}}$$

where $Speedup_{ideal}$ denotes the ideal speedup without consideration of move operations overhead, and $Speedup_{reg}$

represents the real speedup if the communication cost is included. It is clear that the communication overhead will seriously offset the speedup achieved from the extended instructions. On average, this speedup will drop 41% and 32% under the 3-input and 4-input constraints respectively. Therefore, data bandwidth seriously degrades the performance improvement for configurable processors.

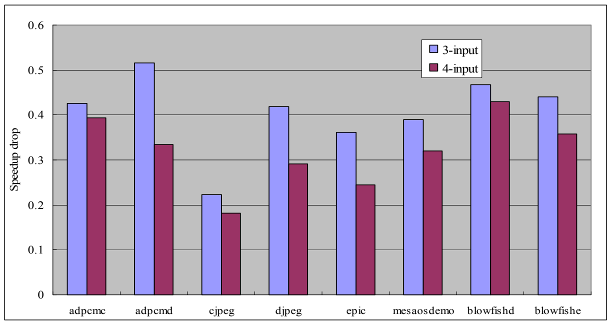


Figure 5. Speedup drop with different input constraints.

3. ARCHITECTURE EXTENSION

3.1 Existing Solutions

Several architectural approaches can be adopted to tackle the speedup degradation caused by the port number limitation. Wider data bandwidth can be achieved by reducing the communication latency or allowing more operands for an instruction. We shall discuss three architectural approaches below.

3.1.1 Dedicated Data Link

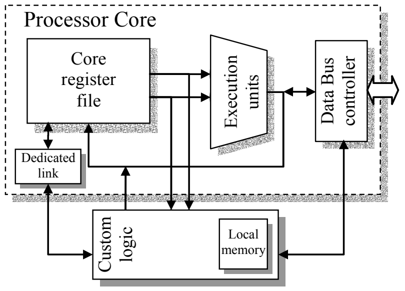


Figure 6. Use dedicated link to reduce communication cost.

Compared to a data bus with potential resource contentions, a dedicated data link can reduce the latency of the communication between processor core and custom logic. Figure 6 shows that a dedicated link can be introduced to the configurable processor to facilitate the communication between the core register file and the custom logic.

The dedicated link approach is employed in Microblaze [25], an embedded processor from Xilinx. A special interface called LocalLink is provided to allow fast and direct access between the Microblaze processor and the custom logic. Although Microblaze does not allow the custom logic to access the processor's register file directly, a data transfer can be performed through LocalLink within only two cycles, which is very fast compared to a system bus. The custom logic which implements critical function kernels can exchange arbitrary volume of data with the processor very efficiently through LocalLink.

However, extra instructions should be introduced to control the dedicated link. For example, in Microblaze instructions PUT and GET are used for this purpose. Since extra CPU cycles are required to accomplish the communication, the latency overhead will be very large if the speedup from hardware acceleration is relatively small.

3.1.2 Multiport Register File

A straightforward method to increase data bandwidth for an instruction is to increase its allowed operand number, but this requires the use of a multiport register file to introduce extra read ports used exclusively by the custom instructions. This allows the custom instruction to increase simultaneous accesses to the core register file. No communication latency will be introduced if the operand number is no more than the number of the register read ports.

However, since the base instruction set is untouched, the extra read ports will be wasted when executing the basic instructions. In addition, adding ports to the register file will have a dramatic impact on the energy and die area of the processor. As pointed out in [18], the area and power consumption of a register file grows cubically with its port number.

Moreover, since the register file is controlled solely by the core processor. In order to access an additional read port of the register file, a custom instruction needs one extra address encoded in its instruction word. This may not be feasible because of the limited instruction word length.

3.1.3 Register File Replication

Register file replication is another technique to increase the data bandwidth. By creating a complete physical copy (or partial copy) of the core register file, the custom instructions can fetch the encoded operands from the original register file and the extra operands from the replicated register file. Chimaera [22] is capable of performing computations that use up to nine input registers by using this approach.

Since the basic instructions cannot utilize the replicated register file, this technique also introduces considerable resource waste in terms of area and power. In addition, this approach enforces a one-to-one correspondence between the registers in core register file and those in replicated register file, and the computation results are always copied to the same corresponding replicated registers. As a result, it leaves very limited opportunities for compiler optimization to further improve the performance.

3.2 Our Approach — Shadow Registers

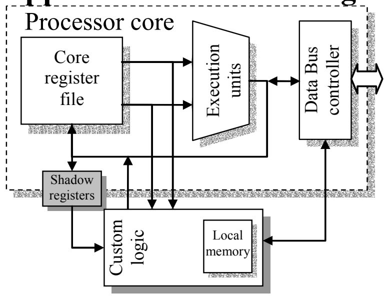


Figure 7. Introducing shadow registers.

To overcome the aforementioned limitations and difficulties, we introduce shadow registers to enhance the configurable processor architecture. Figure 7 shows the block diagram of this architecture, in which the core register file are augmented by an extra set of shadow registers that are conditionally written by the processor in the write-back stage and used only by the custom logic.

3.2.1 *Controlling the Shadow Registers*

An instruction, whether basic or extended, can either skip or forward the result into one of the shadow registers in the write-back stage. The forward/skip option and the address of the target shadow register need to be encoded as additional control bits in the instruction format. Table 2 shows a possible encoding scheme for the extension with three shadow registers, in which two bits are sufficient.

Table 2. An instruction encoding with 3 shadow registers.

Operation	Forward the result to the target shadow register			Skip
Instruction subword	00	01	10	11
Target shadow register ID	0	1	2	-

In the write-back stage, the control of the processor core provides the write-enable and address signals to the shadow registers. While in decode/execution stages, the shadow registers are controlled by the custom logic with read-only privilege. It is obvious that the communication between the processor core and the custom logic is free of communication overhead if the data can be forwarded to the shadow registers.

3.2.2 *Advantages and Limitations*

Since the shadow registers will be mainly used for storing variables with short lifetimes within the basic blocks, the required number of shadow registers is usually much smaller than that of the core register file. Therefore, the implementation tends to be very cost-efficient when compared to the approaches of using extra register ports and register file replication. Except for the shadow registers and the forward path, the other datapath structures can remain the same and only a few control signals need to be added.

For the ISA, since the number of the shadow registers is relatively small (no more than three in general), very few bits (no more than two) need to be encoded. We believe that these control bits can be added without increasing the length of the instruction word as the unused opcodes are usually available (especially in 32-bit instruction format). For example, there exist five reserved bits in NiosII R-type instructions [23], which can be potentially used for the advanced features.

Moreover, we require that a shadow register remain at its proper value during the time a custom instruction reads that register and the time it completes. This should be handled by the compiler so that an active shadow register would not be overwritten by another instruction.

In our ASIP design flow, the compiler will maximize the shadow registers usage by carefully scheduling and register binding. We will investigate an interesting shadow register binding problem in Section 4. The custom logic implementation should then follow the register binding results and obtain desired operands from the correct register addresses.

4. BINDING FOR SHADOW REGISTERS

4.1 Preliminaries

Compiler optimization algorithms are usually performed on the *control data flow graph* (*CDFG*) derived from the program. On the top level of a CDFG, the *control flow graph* consists of a set of basic block nodes and control edges. Each basic block is a *data flow graph* (*DFG*), in which nodes represent basic computations (instruction instances) and edges represent data dependencies. A DFG is essentially a directed acyclic graph (DAG), and we use *G(I, E)* to denote it hereafter. Each node (instruction instance) *i* in *G(I, E)* is associated with a number indicating its execution latency, denoted as *Latency(i)*. For a data edge *e(p, q)*, which has predecessor node *p* and successor node *q*, *p* and *q* are called *e*'s producer and consumer, respectively. In this paper, we will only focus on the shadow register binding problem for data flow graphs, i.e., within the basic block boundary.

For the sake of simplicity, we assume each instruction instance produces only one result in this work, and we assume that static single assignment (SSA) [7] has been performed so that each assignment for a variable has a unique name. Therefore, in a DFG, one node (instruction instance) corresponds to one variable and vice versa. Hereafter, we will not distinguish a node and the variable it produces. A data edge in a DFG actually represents a use of a variable, and a variable may have multiple uses (or data edges) by different consumer nodes. For example, in the DFG of Figure 8, node i_1 produces a variable (also denoted as i_1) consumed by node i_2 and i_4, resulting in two data edges e_1 and e_2, accordingly.

In a scheduled DFG, an instruction *i* is associated with a scheduled time slot *T(i)* indicating its execution order, and the *lifetime of a data edge e(p, q)* is denoted as an interval [*D*(*e*), *U*(*e*)] = [*T*(*p*)+*Latency*(*p*), *T*(*q*)]. The *lifetime of a variable i* is defined as the maximum of the lifetimes of the data edges produced by *i*. For example, in the DFG of Figure 8, suppose Latency(i_1) = 1, the lifetime of the variable i_1 is interval [2, 4], while the lifetimes of the two uses (e_1 and e_2) are [2, 2] and [2, 4] respectively.

4.2 Motivation

Based on the CDFG, our ASIP compiler generates extended instructions and maps the application to the extended instruction set so that every node in the mapped CDFG corresponds to an instruction in the extended instruction set (i.e., basic instructions plus extended instructions).

$i_1 = \ldots;$

$i_2 = ext_1(\ldots, i_1, \ldots);$

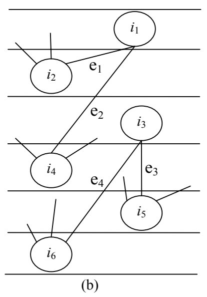
$i_3 = \ldots;$

$i_4 = ext_2(\ldots, i_1, \ldots);$

$i_5 = ext_3(\ldots, i_3, \ldots);$

$i_6 = ext_4(\ldots, i_3, \ldots);$

(a)



(b)

Figure 8. An instruction sequence and its data flow graph.

We assume that the instruction scheduling is done prior to the shadow register binding. If a variable is allocated into the shadow register, all its consumers within an extended instruction can retrieve the value from the shadow register. We classify the data edges into groups so that the edges in each group come from the same producer.

As mentioned earlier, if the instruction set allows N input operands and one output operand, for the extended instructions with more than N inputs, extra data transfer (or move, for short) operations are needed to copy operands from the register file to the local storage in the custom logic. In our proposed architecture, if an operand is already in the shadow register, one move operation can be saved.

Register binding has been extensively studied in both compiler [1] and high-level synthesis [17] domains. Given a set of selected input edges of the extended instructions which have more than N inputs, we construct a compatibility graph for these variables, where each vertex corresponds to a variable, and there is a directed edge (v_i, v_j) between two vertices if and only if their corresponding lifetimes do not overlap and $D(v_i)<D(v_j)$. The variables can be assigned to the shadow register if and only if they are compatible with each other. This formulation can then be reduced to the clique partitioning problem.

However, we observe that it is possible to achieve better solutions by allowing the variables to be replaced in the middle of their lifetimes. For the example in Figure 8, suppose the register file has only two read ports, and all the extended instructions have three input operands. Four move operations will be required without the shadow register. If we keep the variables in the shadow register for their whole lifetimes, only two moves can be saved through the shadow register. Interestingly, one more move can be saved if instruction i_3 commits to the shadow register in cycle 3 and overwrites the result of i_1. Therefore, we have the following observations.

OBSERVATION 1: It is not necessarily optimal to keep a variable in the shadow register for its entire lifetime. This suggests that we should focus on the binding problem for the uses of a variable instead of the variable itself.

OBSERVATION 2: On the other hand, if a variable use is bound to one particular shadow register, it automatically implies that all the previous uses of this variable are also bound into the same shadow register.

```
I1.    a = …;
I2.    b = …;
I3.    c = …;
I4.    d = …;
I5.    e = …;
I6.    … = ext1 ( a, b, c);
I7.    … = ext2 ( d, e, a);
```

Figure 9. Example code sequence for input operand selection.

Another important problem is to determine which input operands should be bound to the shadow registers. Let S be the set of extended instructions with more than N inputs, and M_i denotes the number of inputs for extended instruction i. In order not to waste the bandwidth provided by the core register file, we have another observation as follows.

OBSERVATION 3: For an instruction with M inputs ($M>N$), at most M-N operands (or variable uses) can be bound to the shadow registers.

Therefore, for each extended instruction in this set, there are $C_{M_i}^{M_i-N}$ shadow register candidates. To consider all the extended instructions, the number of the combination grows exponentially. Binding different candidates to the shadow register will lead to different savings. For the example in Figure 9, if only one shadow register is available, two saves can be made if input a is selected for both of the extended instructions. If we select other input operands, only one move can be saved. The optimal solution can be obtained if we search all the combinations. This is unaffordable due to the extremely large search space. In the following subsections, we shall present an efficient algorithm to solve the shadow register binding problem.

4.3 Binding for One Shadow Register

The binding problem for one shadow register can be formulated as follows:

PROBLEM: Binding for one shadow register.

Given a scheduled DFG graph $G(I, E)$ and a shadow register, bind the variables to the register so that the maximum number of move operations can be saved.

To accurately calculate the move reduction, a weighted compatibility graph can be built in the following way. Different from the conventional compatibility graph, each vertex corresponds to a data edges in the original DFG. There is an edge from v_{ei} to v_{ej} if and only if the lifetimes of the corresponding data edges do not overlap and $D(e_i)<D(e_j)$. Each node v_{ei} is assigned a weight which denotes the number of move saves if the variable value is kept in the shadow register until the use time $U(e_i)$. As explained above, if a value is in the shadow register, all the consumers can retrieve the value from it. If we sort the data edges in a group in an ascending order of their use times, the weight equals the index of that edge. For the example in Figure 8, the weight of e_1 and e_2 is one and two respectively. We define the *Cover-Set*(e_i) of a data edge e_i as the set of edges from the same group of e_i, and their use times are earlier than $U(e_i)$.

FACT 1: The input edges of an extended instruction are not compatible with each other.

This is straight-forward because their lifetimes overlap at the end time. Similarly, it is also easy to get the following fact.

FACT 2: The output edges from a node in the DFG are not compatible with each other.

A partially ordered set (POSET) P is a collection of elements with a binary relation $\leftarrow$ defined on $P\times P$ which satisfies reflexive, anti-symmetric, transitive properties [16]. We say that x and y are *related* if we have either $x\leftarrow y$ or $y\leftarrow x$. A *chain* in P is a subset of elements such that any two of them are related. Given a compatibility graph $G_c = (V_c, A_c)$, let POSET $P_c = \{v_1, v_2, ..., v_n\}$ such that P_c contains all the vertices of G_c, and the compatibility relation defined in A_c can be the relation $\leftarrow$ on the elements of P_c. It is easy to show that the compatibility relation is reflexive, anti-symmetric, and transitive. We copy weights of nodes in G_c to the corresponding elements in P_c.

LEMMA: The one shadow register binding problem is equivalent to find a maximum weighted chain in the POSET P_c.

The basic idea of the proof is as follows. The nodes on the chain are compatible with each other, so their corresponding variables can be allocated to the same shadow register. Fact 1 guarantees that at most one input operand for each extended instruction is in the shadow register, so we will not waste shadow registers for input operands which could be retrieved from the core register file without any additional cost. The weight on a node indicates the total number of saves for storing the value in the shadow register until the end time. Fact 2 implies that a variable could only be bound to the shadow register at most once. So the maximum weighted chain corresponds to a register binding with maximum move saves.

Since the POSET can be constructed in $O(|V'|^2)$, the maximum weight chain can be solved in $O(|V'| + |E'|)$, we can directly derive the following theorem.

THEOREM: One shadow register binding problem can be solved optimally in time $O(|V'|^2)$.

Another nice property of our algorithm is that the input bound to the shadow register is selected simultaneously in the binding process.

4.4 Extension to *K* Shadow Registers

The algorithm can be easily extended to a heuristic that handles K shadow registers by iteratively solving the one shadow register binding problem. After one maximum weighted chain is found, the elements in the chain are removed and the corresponding data edges are marked. For a marked edge e_i, the edges in the Cover-Set(e_i) should also be marked because their value is already in the shadow registers. We then examine the extended instructions. If the number of unmarked input edges is no more than N, no additional move operations are needed for this instruction, and all of their input edges should be removed from the compatibility graph. After this iteration, we repeat the process until the graph becomes empty or no shadow registers can be allocated.[1]

4.5 Experimental Results

We implemented our algorithms in a C++/Unix environment. A new step called *shadow register binding* is performed after application mapping in our compilation flow. The mapped applications with shadow register binding are fed into SimpleScalar to measure the performance improvement.

By introducing the shadow register, the number of move operations will be effectively reduced. Figure 10 shows the speedup with different numbers of shadow registers and different input constraints. Approximately 89% of the performance gap can be closed with three shadow registers for 3-input constraint. Intuitively, the more shadow registers are provided, the more speedup can be achieved. Since our current algorithm only performs the shadow register binding within the basic block boundary, the values that are produced outside the basic block cannot be put into the shadow registers. This is the main reason that some benchmarks cannot get further speedup even with additional shadow registers. On average, our proposed shadow register and register allocation algorithm close 72% of the performance gap for the 3-input and 4-input constraints.

[1] Note that this extension does not guarantee the optimal K-shadow-register binding by iteratively solving the sub-problems for single shadow registers. Readers may bring up another promising approach based on the k-cofamily [9] formulation, which has been successfully applied to the register allocation problem [4] under the context of behavioral synthesis. However, our study shows that the k-cofamily-based algorithm is not directly applicable since it cannot satisfy all the three observations mentioned in Section 4.2, which altogether constitute a necessary optimality condition for the shadow register binding problem. Due to the lack of in-depth complexity analysis, we currently resort to the iterative heuristic algorithm which is very efficient in runtime with reasonable solution quality.

5. CONCLUSION AND FUTURE WORK

Data bandwidth problem is significantly limiting the performance of application-specific instruction set processors. In this paper, we provide a quantitative analysis of the data bandwidth problem and propose to use the shadow register as a novel low-cost architectural extension to mitigate this limitation. We also formulate a new shadow register binding problem and present an efficient algorithm to solve the problem. The application of our approach results in a promising performance improvement.

In this experiment, we apply the compilation on the binary code. Because memory operations are not allowed in the extended instructions, some extended instruction generation opportunities are lost due to the spilling and loading temporary values. The original register allocation and instruction scheduling also limit the application of our compilation and shadow register binding. In the future work, we will develop the ASIP compilation tool on the source code level. Global shadow register allocation algorithm will also be investigated to further mitigate the bandwidth limitation.

ACKNOWLEDGMENT

This research is partially funded by MARCO/DARPA Gigascale Silicon Research Center (GSRC), National Science Foundation under award CCR-0096383, and grants from Altera Corporation and Xilinx, Inc. under the California MICRO program.

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[2] K. Atasu, L. Pozzi, and P. Ienne, "Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints," in *Proc. 40th Design Automation Conference*, pp. 256-261, Jun. 2003.

[3] D. Burger, T. Austin, and S. Bennett, "Evaluating Future Microprocessors: The SimpleScalar Toolset," *Technical Report, CS-TR96-1308*, Univ. of Wisconsin - Madison, 1996.

[4] D. Chen and J. Cong, "Register Binding and Port Assignment for Multiplexer Optimization," in *Proc. the Asia Pacific Design Automation Conference*, pp. 68-73, Jan. 2004.

[5] K. Compton, S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Computing Surveys (CSUR),* vol. 34(2), pp. 171-210, Jun. 2002.

[6] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-Specific Instruction Generation for Configurable Processor Architectures," in *Proc. ACM International Symposium on Field-Programmable Gate Arrays*, pp. 183-189, Feb. 2004.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadek, "An Efficient Method of Computing Static Single Assignment," in *Proc. ACM Symposium on Principles of Programming Languages*, pp. 25-35, Jan. 1989.

[8] D. Fischer, J. Teich, M. Thies, and R. Weper, "Efficient architecture/compiler co-exploration for ASIPs," in *Proc.*

International Conference on Compilers, Architecture, and Synthesis for Embedded System, pp. 27-34, Oct. 2002.

[9] C. Greene and D. Kleitman, "The Structure of Sperner K-Family," *J. Combinatorial Theory*, Ser. A, vol. 20, pp. 80-88, 1976.

[10] M. R. Guthaus et al., "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *IEEE 4th Workshop on Workload Characterization*, Dec. 2001.

[11] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," in *Proc. 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 24-33, Apr. 1997.

[12] P. Ienne, L. Pozzi, and M. Vuletic, "On the Limits of Processor Specialisation by Mapping Dataflow Sections on Ad-hoc Functional Units," *Technical Report 01/376, Swiss Federal Institute of Technology Lausanne, Computer Science Department*, Dec. 2001.

[13] R. Kastner, A. Kaplan, S. Ogrenci Memik, and E. Bozorgzaden, "Instruction Generation for Hybrid Reconfigurable Systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 7, pp. 605-627, Oct. 2002.

[14] K. Keutzer, S. Malik, and A. R. Newton, "From ASIC to ASIP: The Next Design Discontinuity," in *Proc. International Conference on Computer Design*, pp. 84-90, Sept. 2002.

[15] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A Tool for Evaluating Multimedia and Communications Systems," in *Proc. 30th International Symposium on Microarchitecture*, pp. 330-335, Dec. 1997.

[16] C. L. Liu, *Elements of Discrete Mathematics*, McGraw-Hill, 1977.

[17] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[18] S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens, "Register Organization for Media Processing," in *Proc. Sixth International Symposium on High-Performance Computer Architecture*, pp. 375-386, Jan. 2000.

[19] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "Synthesis of Custom Processors based on Extensible Platforms," in *Proc. International Conference on Computer-Aided Design*, pp. 256-261, Nov. 2002.

[20] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "A Scalable Application-Specific Processor Synthesis Methodology," in *Proc. International Conference on Computer-Aided Design*, pp. 283-290, Nov. 2003.

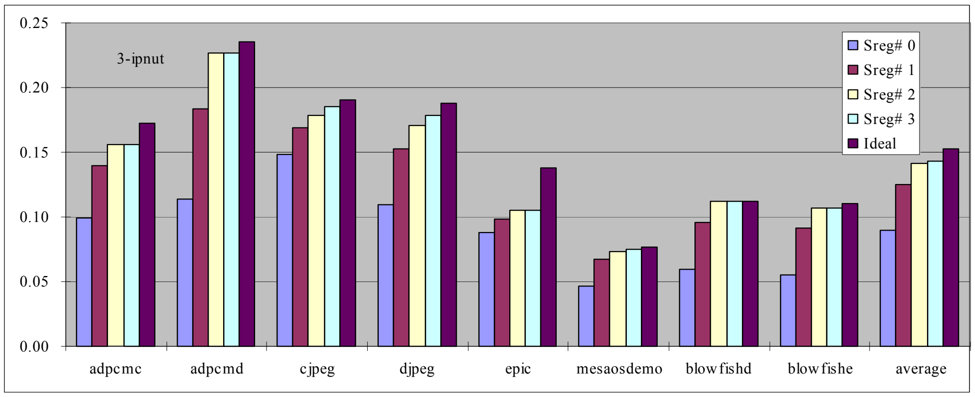
[21] R. D. Wittig and P. Chow, "OneChip: An FPGA Processor with Reconfigurable Logic," in Proc. *4th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 126-135, March 1996.

[22] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit," in *Proc. 27th Annual International Symposium on Computer Architecture*, pp. 225-235, Jun. 2000.

[23] Altera Corp., http://www.altera.com.

[24] Tensilica Inc., http://www.tensilica.com.

[25] Xilinx Inc., http://www.xilinx.com.



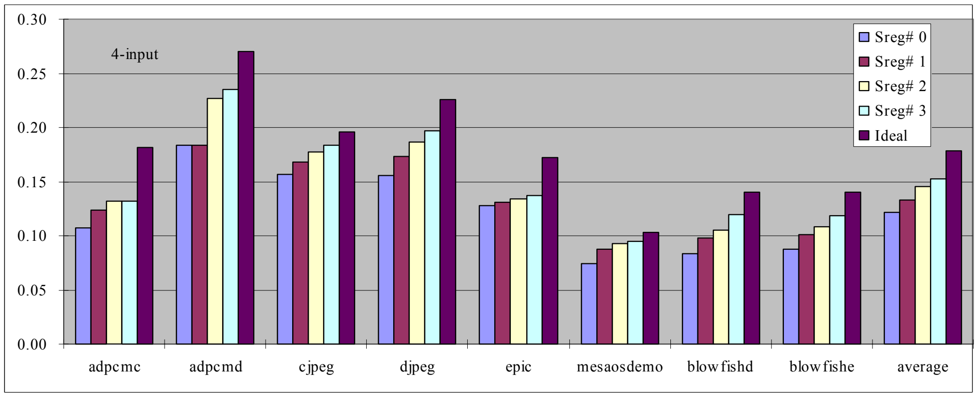


Figure 10. Speedup under different number of shadow registers.