Architecture and Compilation for Data Bandwidth Improvement in Configurable Embedded Processors

Jason Cong, Guoling Han, Zhiru Zhang
Computer Science Department
University of California, Los Angeles, CA 90095
{cong, leohgl, zhiruz}@cs.ucla.edu

ABSTRACT

Many commercially available embedded processors are capable of extending their base instruction set for a specific domain of applications. While steady progress has been made in the tools and methodologies of automatic instruction set extension for configurable processors, recent study has shown that the limited data bandwidth available in the core processor (e.g., the number of simultaneous accesses to the register file) becomes a serious performance bottleneck.

In this paper we propose a new low-cost architectural extension and associated compilation techniques to address the data bandwidth problem. Specifically, we embed a single control bit in the instruction op-codes to selectively copy the execution results to a set of hash-mapped shadow registers in the write-back stage. This can efficiently reduce the communication overhead due to data transfers between the core processor and the custom logic. We also present a novel simultaneous global shadow register binding with a hash function generation algorithm to take full advantage of the extension. The application of our approach leads to a nearly-optimal performance speedup (within 2% of the ideal speedup).

1. INTRODUCTION

Application-specific instruction-set processors (ASIPs) 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, an ASIP has the capability to extend the instruction set of the base architecture with a set of application-specific instructions implemented by custom hardware resources. These hardware resources can be either runtime reconfigurable functional units, or pre-synthesized circuits. The recent emergence of many commercially available embedded processors with both configurability and extensibility (e.g., Altera Nios/NiosII [17], Tensilica Xtensa V/LX [18], Xilinx MicroBlaze [19], etc.) testifies to the benefit of this approach.

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 manage by manual designs, and is further complicated by various micro-architectural constraints. This has motivated a large body of recent research to address the automatic instruction set extension problem.

Most of the existing approaches attempt to discover the candidate extended instructions by identifying common patterns in the data flow graph of the given application, and then to select an appropriate subset of the candidate instructions to maximize performance under certain architectural constraints (e.g., the number of input and output operands, area constraints, etc.). Kastner et al. [12] proposes a simultaneous template generation and matching method that constructs the candidate instruction set by clustering the nodes based on edge-type frequencies. Sun et al. [16] enumerates the possible extended instructions by repeatedly combining the smaller templates to form the larger templates. A comprehensive priority function is computed to rank and prune the candidate instructions. The method was recently extended by [6] to speed up the exploration process. Atasu et al. [4] imposes further constraints on the instruction topology and performs a branch-and-bound algorithm on a binary tree to determine whether or not to include a node into the candidate instruction.

Although the existing techniques are efficient in identifying the promising candidate instructions, [11] points out that most of the speedup (about 60%) comes from the cluster with more than two input operands. 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 [16] 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 incurred because of these data transfers between the core processor and the custom logic can significantly offset the gain from forming a large cluster.

A quantitative analysis of the data bandwidth limitation is given in [8], and the problem is directly addressed by augmenting the processor's register file with an extra set of registers that are written transparently by the processor and used by the custom instructions. This avoids explicit move instructions from the core register file to the custom logic. Experiments show that a small number of shadow registers is sufficient to compensate a major portion of the performance degradation caused by the port number limitation. However, additional bits (two to three bits) need to be added in the instruction format, which is difficult to be encoded without increasing the instruction word length. A shadow register binding algorithm is also proposed as the associated compilation technique in [8]. This algorithm limits its scope within the basic block boundary.

Our contributions in this paper are twofold. First, we propose a new low-cost architectural extension called hash-mapped shadow registers, which enables and significantly enhances the shadow-register-based scheme using a single control bit. Second, we present a simultaneous global shadow register binding with a hash function generation algorithm which fully exploits the benefits of shadow registers across the basic block boundaries.

The remainder of the paper is structured as follows. We first present our architectural extension in Section 2. The algorithmic details for solving the global shadow register binding problem together with the hash function construction problem are

described in Section 3. Experimental results are shown in Section 4, followed by conclusions in Section 5.

2. ARCHITECTURE EXTENSION

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. Under this processor model, a custom instruction follows the same instruction format and execution rules, which include: (1) The number of operands and results of a custom instruction are pre-determined by the base 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 be retained.

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 is relaxed, higher performance speedup can be achieved by clustering more operations in one custom instruction to exploit more parallelism. 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, which may take multiple CPU cycles. This communication overhead will significantly offset the performance gain experienced by using the custom instruction.

2.2 Existing Solutions

Several architectural approaches can be adopted to tackle the speedup degradation caused by the port number limitation. We shall discuss three existing approaches below.

Multiport Register File: A straightforward method to increase data bandwidth for an instruction is the use of a multiport register file to introduce extra read ports. This allows the custom instruction to increase simultaneous accesses to the core register file. However, adding ports to the register file will have a dramatic impact on the energy and die area of the processor. As pointed out in [15], the area and power consumption of a register file grows cubically with its port number.

Register File Replication: Register file replication is another technique used to increase 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. By using this approach, Chimaera [10] is capable of performing computations that use up to nine input registers. The reconfigurable logic is given direct read access to a subset of the registers in the processor by creating an extra register file which contains copies of those registers values, thus allowing complicated computations to be performed.

Since the basic instructions cannot use the replicated register file, the complete register duplication approach will introduce considerable resource waste in terms of area and power. For the partial duplication approach, it enforces a one-to-one correspondence between a subset of registers in the core register file and those in the replicated register file, and the computation results are always copied to the replicated registers. This leaves very limited opportunities for compiler optimization to further improve performance.

Shadow Registers with Multiple Control Bits: Shadow registers have been recently proposed [8] as an architectural extension to overcome the aforementioned limitations and difficulties. Figure 1 shows the block diagram of this architecture, in which the core register file is 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.

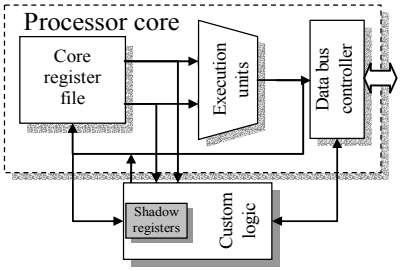

Figure 1. Shadow registers with multiple control bits.

Any instruction (basic or extended) can either skip or copy the computation result into one of the shadow registers in the write-back stage. The copy/skip option and the address of the target shadow register need to be encoded as additional control bits in the instruction format. Table 1 shows one possible encoding for the extension with three shadow registers, in which two bits are needed.

Table 1. An instruction encoding for three shadow registers.

Operation	Copy to shadow register			Skip
Instruction subword	00	01	10	11
Target shadow	0	1	2	-

Since the data is copied to the shadow registers transparently during the write-back stage, the communication overhead between the processor core and the custom logic can be greatly reduced. Moreover, under the above encoding scheme, each shadow register can be the physical copy of any register in the core register file, which creates a great amount of freedom and opportunities for the compiler optimization. Experiments in [8] have shown that a small number (typically less than four) of shadow registers is sufficient to compensate a major portion of the performance degradation caused by the port number limitation.

However, additional bits ($\lfloor \log_2 K \rfloor$+1 bits for K shadow registers) are required in the instruction format. Since the unused opcodes are usually limited in the common instruction format, it is not trivial to encode two or three extra bits without increasing the word length. In fact, we need to represent four distinct versions of one instruction to accommodate two extra control bits and eight versions for three bits, which is not practical in general.

2.3 Proposed Approach — Shadow Registers with Single Control Bit

Note that the shadow registers essentially act as a tiny cache located in the customer logic to temporarily store the recent computation results from the core processor. In this sense, the multiple-bit controlled shadow register set resembles a fully associative cache since each core register can be mapped to any entry of shadow register set. This full associativity requires the presence of multiple control bits in the instruction word to

identify the target shadow register. In this subsection, we propose to use a hash-mapped shadow register scheme to significantly reduce control bit overhead.

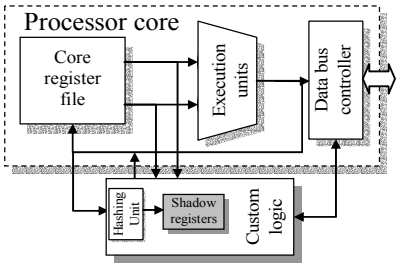

Figure 2. Shadow registers with single control bit.

2.3.1 Hash-Mapped Shadow Registers Scheme

Figure 2 shows the block diagram of our proposed enhancement. In this scheme, every instruction can also choose to either skip or copy the result into the shadow registers during the write-back stage. The key difference is that only the copy/skip option needs to be encoded as an additional control bit in the instruction format.

The actual shadow register that will be written is determined by a pre-specified many-to-one hash function. Namely, the execution result to register $R[i]$ in the core register file will be conditionally copied to register $SR[j]$ in the shadow register set where $j = hash(i)$. For example, one simple hashing scheme can be directed mapping where $j = i \bmod K$, assuming that we have K shadow registers. In general, the particular hash function can be performed by the hashing unit which resides in the custom logic using a fast and cheap lookup table, and reconfigured for different applications or different domains of applications.

2.3.2 Advantages and Limitations

Since the shadow registers will be mainly used for storing the source operands of the custom instructions, the required number of shadow registers is generally much less than that of the core register file. Except for a small amount of extra interconnect and control logic introduced by the conditional copy path, the base datapath will remain the same. Therefore, the implementation tends to be very cost-efficient when compared to the prior approaches of using the multiport register file and register file replication.

Note that the target shadow register address is determined by the hashing unit instead of being explicitly encoded in the instruction word. This allows more shadow registers (greater than three) without the penalty of increasing the number of control bits. More importantly, we believe that it is much easier to add (or encode) one single additional control bit without increasing the length of the instruction word. For example, in NiosII [17] R-type instructions there are several reserved bits that can be potentially used for advanced features. For the I-type and J-type instructions, there are 42 encodings specified for the 6-bit OP code field. This leaves more than 20 encodings unused, which are sufficient to accommodate the second versions of those particular instructions (e.g., arithmetic and load instructions) that can forward their results to the shadow registers. Clearly, our proposed single-bit controlled shadow register approach provides a much more viable solution compared to the original multi-control-bit shadow registers approach reviewed in Section 2.2.

One potential limitation of the hash-mapped shadow registers approach is that a predetermined many-to-one correspondence is enforced between the core registers and the shadow registers, which may restrict the opportunities for compiler optimization to further improve performance. However, we believe this limitation is minor because more shadow registers can be allocated to mitigate the problem. Essentially, we can trade off the associativity of the shadow register set for the number of entries.

3. SIMULTANEOUS GLOBAL SHADOW REGISTER BINDING WITH HASH FUNCTION GENERATION

Shadow registers should be also considered during the compilation process to fully exploit the benefits of our proposed architectural extension. First of all, the compiler has to guarantee the integrity and the correctness of the program, e.g., it has to ensure that an active shadow register would not be overwritten during the execution of a custom instruction. Moreover, optimization techniques are needed to intelligently assign or bind the variables to the appropriate shadow registers. In [8], a shadow register binding problem was introduced as a special case of register allocation problem to maximize the usage of the shadow registers. Unfortunately, the proposed algorithm limits itself to a data flow graph only (i.e., within the basic block boundary).

In this section we first present a global solution that performs shadow register binding across the whole control data flow graph using the predetermined hash function described below in subsection 3.1. We then show that the hash function generation can be carried out simultaneously with the shadow register binding in subsection 3.2.

3.1 Global Shadow Register Binding Under Prescribed Hash Function

3.1.1 Preliminaries

Compiler optimization algorithms are usually performed on the control data flow graph (*CDFG*) derived from a 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 (or instruction instances) and edges represent data dependencies. Throughout this section, we assume that given the profiling information, the ASIP compiler has generated extended instructions and mapped the application so that every node in the mapped CDFG corresponds to an instruction in the extended instruction set (including basic instructions and custom instructions). We also assume that each instruction produces at most one result, and the instruction scheduling and register allocation for the core registers have already been performed. Thus we know the shadow register to which an operand can be mapped with the given hash function.

Our task is to appropriately assign the source operands of the custom instructions to the available shadow registers so that the potential communication overhead (in terms of the number of move operations) can be minimized. Specifically, the problem is solved in three steps: (i) We first perform a depth-first search to produce a linear order of all the instructions; (ii) Then we derive the live intervals for each shadow register candidate (i.e., data use); (iii) Based on the live intervals, we construct a compatibility graph and bind each shadow register independently.

3.1.2 Linear Ordering

We adapt a linear scan register allocation technique to compute a global linear order of the instructions and derive the live intervals for the shadow register candidates in the CDFG. Linear scan register allocation was proposed by Poletto and Sarkar in [14]. It is a very simple and efficient algorithm that runs up to seven times faster than a fast graph coloring allocator and produces relatively high-quality code.

We employ the depth-first method introduced in [1] to order the basic blocks. The final linear ordering is the reverse of the order in which we last visit the nodes in the preorder traversal of the graph. Figure 3 shows an example CDFG annotated with the linear numbering for each basic block. The complete sequence of basic blocks visited as we traverse the CDFG is 1, 2, 3, 4, 6, 7, 6, 4, 5, 4, 3, 2, 1. In this list, we mark the last occurrence of each number to get 1, 2, 3, 4, 6, 7, 6, 4, 5, 4, 3, 2, 1, which is the final ordering of the basic blocks. Additionally, the instructions inside one basic block are ordered according to the instruction scheduling. By combining these two numberings, we obtain a global sequence number (*SN*) for each instruction in the CDFG. In fact, the correctness of the algorithm does not depend on the ordering method. However, it may influence the quality of register binding.

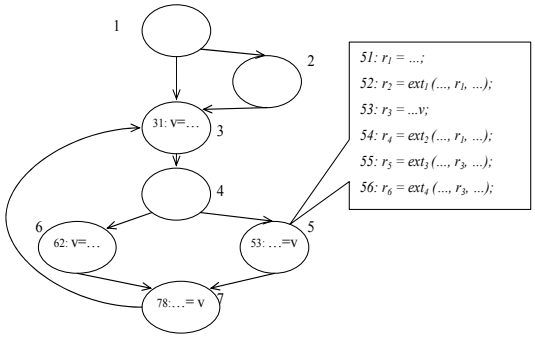

Figure 3. A CDFG example with linear order.

3.1.3 Live Interval Generation

Motivation

As mentioned earlier, if the input number of a custom instructions exceeds the available read port count, 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.

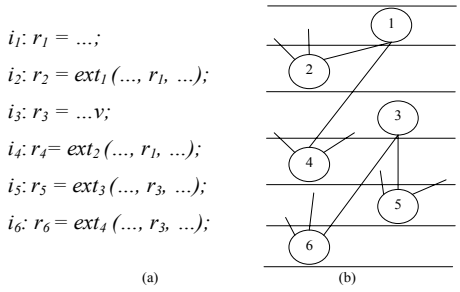

Figure 4. An instruction sequence and its data flow graph.

In addition, it is not necessarily optimal to keep a variable in the shadow register for its every use as pointed out in [8]. Figure 4 shows the data flow graph of the basic block 5 in Figure 3. Suppose the register file has only two read ports, and all the extended instructions have three input operands, then four move operations will be required without the shadow register. Assuming that there is one shadow register available, only two moves can be saved through the shadow register if we keep the variables in the shadow register for their entire lifetime. Interestingly, one more move can be saved if instruction i_3 commits to the shadow register and overwrites the result of i_1. Therefore, to achieve the maximum number of move operation saves, we should allow a value to be replaced in the middle of its lifetime. This motivates us to define the life intervals with respect to each data use of a variable instead of the variable itself.

Definition of Live Intervals

To derive the live interval of a data use r, we need to consider the following three cases: (1) If variable r is defined (or assigned) within the same basic block of the subject use, and the definition is before the use, then the live interval is $[s, t]$ where the SN of the definition instruction is s and the SN of the use instruction is t; (2) If all the definitions of variable r locate in the preceding basic blocks (precedence relationship is defined in terms of the linear order), then the live interval is $[\min(s_i), t]$ where $\min(s_i)$ denotes the SN of the definition instruction with the smallest numbering in the linear order; (3) If variable r has one definition after the subject use, it implies that r is assigned and used in a loop. In this case, we need to traverse the loop once, extending the live interval to $[i^*, j^*]$ where i^* and j^* denote the smallest and largest SNs of any instructions in this loop.

For the example in Figure 3, the use v at basic block 7 has two definitions in basic blocks 3 and 6, respectively. The live interval for use v is [31, 78]. We should notice that the life interval is a conservative approximation. Some subranges within $[i, j]$ where variables are not live may also be included and are underutilized. However, the correctness of the program is not affected.

3.1.4 Binding for One Shadow Register

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

One-shadow-register binding problem: Given a shadow register *sr*, a hash function *h*, and an interval set *S* in which each interval will be hash-mapped to *sr*, select a subset of non-overlapping live intervals in *S* and bind them to *sr* so that the maximum number of move operations can be saved.

To accurately calculate the move reduction, a weighted compatibility graph $G(V', E')$ can be built in the following way. Different from the conventional compatibility graph, each vertex v corresponds to an interval in *S*. We create an edge from v (with interval $[i, j]$) to v' (with interval $[i', j']$) if and only if $j<i'$. Each vertex is assigned a weight which denotes the number of move saves if the corresponding variable is held in the shadow register until the end time.

FACT 1: The live intervals of the output edges from the same instruction are not compatible with each other.

This is straightforward because their live intervals must overlap at this instruction.

FACT 2: If a data use at instruction i can retrieve the value from the shadow register, the value is also available for other uses along all the reverse paths from the instruction i to the definitions.

Based on this fact, the weight of an interval can be calculated as the sum of move reductions for each use along the reverse paths. In particular, we call these uses are covered by the interval. In Figure 3, suppose instruction 53 and 78 execute 10 and 20 times respectively based on the profiling, then the weight of interval [31, 78] for use v is 30.

LEMMA 1: The weighted compatibility graph is acyclic.

LEMMA 2: The one shadow register binding problem is equivalent to finding a maximum weighted chain in the compatibility graph.

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. The weight on a node indicates the number of saves for storing the value in the shadow register until the end time. Fact 1 implies that a variable, at most, could only be bound to the shadow register once. So the maximum weighted chain corresponds to a register binding with maximum move saves.

Since the interval graphs can be constructed in $O(|V'|^2)$, the maximum weighted 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 $O(|V'|^2)$.

3.1.5 Extension to K Shadow Registers

Recall that the hashing unit determines the shadow register to which a variable can be mapped. Since the hash function is a many-to-one function, each variable can only be mapped to one shadow register. This implies an interesting property wherein each shadow register can be allocated independently. Specifically, for each individual shadow register we can group the corresponding variables (or live intervals) together and perform the one-shadow-register binding algorithm on this group without being interfered with the binding of other hash-mapped shadow registers. Therefore, the algorithm can be easily extended to handle K shadow registers by iteratively solving the one shadow register binding problem.

3.2 Hash Function Generation with Shadow Register Binding

The choice of the hash function implemented in the hashing unit also affects the solution quality. If we revisit the example shown in Figure 4 under the assumption that two shadow registers are available, only two moves can be saved when the hash function accidentally maps core registers r_1 and r_3 to the same shadow register. On the other hand, four moves will be saved if r_1 and r_3 are mapped to different slots. This clearly suggests that hash function generation and shadow register binding are interdependent, and both should be considered together at the same time. In this subsection we present an effective algorithm that constructs the hash function simultaneously with the shadow register binding.

3.2.1 Multi-Way Set Partitioning Formulation

We formalize our hash function generation problem as follows.

Hash function generation (*HSG*) problem: Given a set of core registers $R = \{r_0, \ldots, r_{N-1}\}$, a set of shadow registers $SR = \{sr_0, \ldots, sr_{K-1}\}$, and an objective function f, find a many-to-one function h: $R \rightarrow SR$ so that $f(h)$ is maximized.

In particular, our goal is to generate a best possible hash function h to facilitate the global shadow register binding algorithm so that the maximum number of moves can be saved. Therefore, we evaluate the objective function f by solving the K-shadow-register binding problem described in the preceding section. To be more concrete, for each available shadow register sr_i, we first find all the live intervals whose core register indices are hash-mapped to sr_i. After the live interval set I_i is found, we construct the interval graph G_i for I_i and solve the one-shadow-register binding problem on G_i. We record the result with $w(I_i)$ which denotes the total weight on the maximum weighed chain in G_i (i.e., the number of moves that can be saved for I_i). Once we independently bind all the K shadow registers, the final objective function can be computed by $\sum_{i=0}^{K-1} w(I_i)$.

Since we are searching for a many-to-one function, any register in R can be mapped to one and only one shadow register in SR. Therefore, the desired hash function essentially partitions the given set of core registers into K subsets and labels each subset with a shadow register index. The goal is to maximize the performance gain by saving as many moves as possible. With the above observation, we notice that the hash function generation problem can be polynomially reduced to the well-known multi-way set partitioning problem as described as follows.

Multi-way set partitioning (*MSP*) problem: Given a set of modules $M = \{m_0, \ldots, m_{N-1}\}$, an integer value $2 \le K \le |M|$, and an objective function g, partition V into K disjoint clusters $P^K = \{C_0, C_1, \ldots, C_{K-1}\}$ so that every $m_i \in M$ belongs to exactly one cluster in P^K and $g(P^K)$ is maximized. Specifically, we define our partitioning objective function to be $\sum_{i=0}^{K-1} w(C_i)$ where $w(C_i)$ denotes a weighting function on cluster C_i.

A polynomial reduction from HSG to MSP can be obtained by constructing the module set M in the MSP problem from the set R in the HSG problem and setting the function f to be the partitioning objectives in MSP. Then we can show that an optimal partition uniquely determines an optimal hash function. For example, if $P^2 = \{C_0, C_1\}$ is an optimal 2-way MSP solution on $M = \{m_0, m_1, m_2, m_3\}$ where $C_0 = \{m_0, m_3\}$ and $C_1 = \{m_1, m_2\}$, then it is straightforward to identify that the corresponding hash function h of HSG (with four core registers and two shadow registers) is defined by $h(0) = h(3) = 0$ and $h(1) = h(2) = 1$.

Although the problem of computing an optimal multi-way partitioning is NP-hard in general, because it is so critical to many application areas (especially in VLSI CAD domain), a number of efficient heuristics have been proposed. A comprehensive survey can be found in [3] which provides detailed descriptions and comparisons of various partitioning algorithms. However, most of the existing techniques focus on the min-cut graph (or hypergraph) partitioning problem, which are not directly applicable to our multi-way set partitioning problem with specialized objectives. In this paper, we adapt a two-phase approach proposed in [2] to the HSG problem. First, we apply an efficient heuristic to generate a linear permutation sequence on the core registers set R. Second, we optimally solve a one-dimensional K-way partitioning problem through dynamic programming on the given linear sequence. Once the partitioning problem is resolved, the hash function can be directly constructed together with the global shadow register binding solution.

3.2.2 Core Register Ordering

In the core register ordering phase, our goal is to find a linear permutation on R using a fast heuristic which consists of the following three major steps:

(1) We first compute the total weight of those live intervals associated with each individual core register.

(2) Then we sort the registers in non-decreasing order according to those weights.

(3) Given the sorted list $R^* = \{r^*_0, r^*_1, \ldots, r^*_{N-1}\}$, we begin with first K highest weighted registers, i.e., $r^*_0, r^*_1, \ldots r^*_{K-1}$ and evenly distribute them into the open positions $0, \lfloor N/K \rfloor, \ldots, (K\text{-}1)\cdot\lfloor N/K \rfloor$. Then we distribute the second K highest weighted registers to the open positions $1, \lfloor N/K \rfloor+1, \ldots, (K\text{-}1)\cdot\lfloor N/K \rfloor+1$, and repeat until all core registers are positioned. To be more precise, we construct a linear permutation Π of R so that the register in R^* with index i is moved to position $(i \bmod K)\cdot K + \lceil i/K \rceil$ where $0 \le i \le N\text{-}1$. Figure 5 illustrates the permutation generation process. Suppose we have six registers in a sorted order R^* (i.e., $N = 6$) and $K = 2$, then we will map registers $r^*_0, r^*_1, r^*_2, r^*_3, r^*_4, r^*_5$ to positions 0, 3, 1, 4, 2, 5, respectively.

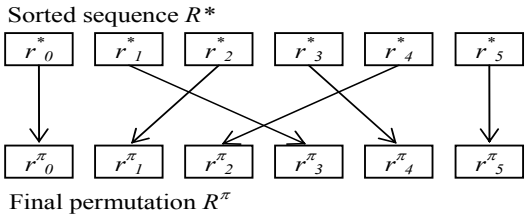

Figure 5. Permutation generation.

We believe the above heuristic suits our purposes because it encapsulates the following intuitions: (i) If two core registers both have large weight (i.e., they are both associated with a large set of live intervals), then there is high possibility that their live intervals frequently overlap and incur many hash conflict (or shadow register contentions) due to the potential shadow register contentions. Hence, these two registers would intuitively repulse each other during the partitioning and they should be spaced out as much as possible in the permutation sequence; (ii) Otherwise, two core registers would attract each other and they should be closed by in the permutation sequence.

Since we space out every two registers with similar high weights by $\lfloor N/K \rfloor$ distance, it is unlikely that they would be placed in the same cluster in a one-dimensional K-way partitioning which will be described in the next subsection.

3.2.3 Optimal One-dimensional K-way Partitioning

Given a linear core register permutation, we solve a one-dimensional K-way partitioning problem to determine the appropriate cluster for each core register.

One-dimensional *K*-way partitioning (*1D-KP*) problem: Given a permutation Π: $R \rightarrow R$, find a K-way set partitioning $P^K_{[0..N-1]} = \{C_0, C_1, \ldots, C_{K-1}\}$ in which each cluster can be uniquely denoted by $C_{[i,j]} = \{r^\pi_i, r^\pi_{i+1}, \ldots, r^\pi_j\}$ where r^π_i represents the ith register in the permuted linear sequence π of R.

Note that we require each cluster of the partitioning to be a contiguous slice of the permutation sequence. Hence, each cluster can be written as $C_{[i, jj}$ which denotes that the cluster begins with ith register and ends with jth register according to the given permuted order.

For such an objective $g(P^K_{[0..N-1]}) = \sum_{i=0}^{K-1} w(C_i)$, the *principle of optimality* holds for the 1D-KP problem, i.e., an optimal solution to any instance is made up of optimal solutions to its sub-instances. Therefore the 1D-KP problem can be optimally solved by dynamic programming. The recurrence function can be expressed as $g(P^{k'}_{[s..t]}) = \max_{i=s}^{t} \{w(C_{[s,i]}) + g(P^{k'-1}_{[i+1,t]})\}$ where $2 \le k' \le K$ and $0 \le s \le t \le N\text{-}1$. The 1D-KP problem has $O(N^2T(w) + KN^3)$ complexity, where $T(w)$ is the time complexity of computing the cluster weighting. As discussed earlier in Section 3.1.4, $T(w)$ is $O(|V'|^2)$ where $|V'|$ represents the size of the interval graph. In fact, both N (usually no greater than 32) and K (typically no greater than 8) are relatively small constants. This helps to significantly reduce the time complexity and makes the algorithm much more scalable.

Since the shadow register binding problems are implicitly solved during the cluster weighting evaluations, we can directly retrieve the global shadow register binding solution when the final partition (or hash function) is determined. Therefore, we solve the hash function generation problem simultaneously with the shadow register binding problem.

4. EXPERIMENTAL RESULTS

In our experiment the cycle accurate simulation tool, SimpleScalar [5], is used to estimate the performance of the processor and the impact of communication cost. To obtain a quick evaluation of data bandwidth limitation, the ASIP compilation is applied on the compiled binary code of the benchmarks.

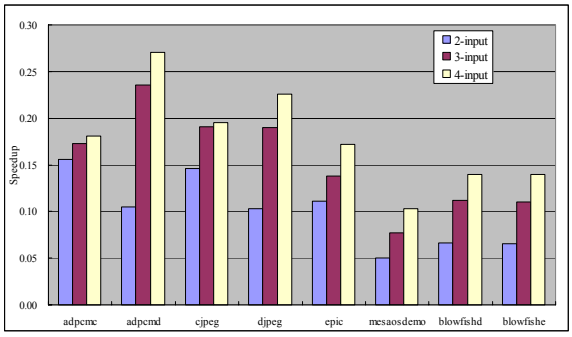

Figure 6. Ideal speedup under different input constraints.

Based on the execution trace generated by SimpleScalar, the CDFG generator constructs the control data flow graph. Under the given micro-architectural constraints (e.g., the number of inputs and outputs), the ASIP compilation problem is solved in three steps based on [7]. The first step, called pattern generation, enumerates all candidate patterns from a given control data flow graph through the cut enumeration technique. These patterns (i.e., candidate custom instructions) are generated within the basic block boundary. Memory operations are not allowed in any extended instruction. 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. 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. Another step, called global 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.

In this work we modeled a single issue, in-order RISC configurable processor which is similar to the Altera Nios/NiosII. We assume the latencies of ALU and multiplier are one and three cycles respectively. The core register file has two read ports and one write port. The detailed machine configuration can be found in [8]. We use the C programs from Mediabench [13] and Mibench [9] suites.

Figure 6 shows the ideal speedup for each benchmark under different input size constraints. 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 indicate that we can achieve 10%, 15%, and 18% speedup on average with the 2-input, 3-input and 4-input constraints, 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 optimistically assume that the move operation needs only one clock cycle. Figure 7 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. On average, this speedup will drop 41% and 32% under the 3-input and 4-input constraints respectively. Obviously, data bandwidth problem seriously degrades the performance improvement for configurable processors.

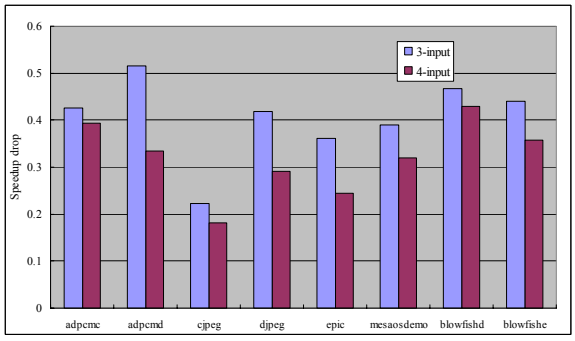

Figure 7. Speedup drop with different input constraints.

By introducing the shadow registers, the number of move operations will be effectively reduced. In our experiment, we apply the simultaneous shadow register binding with hash function generation to allocate the shadow registers. As mentioned earlier, the predetermined many-to-one correspondence enforced by the hash function may cause more conflicts and restrict the opportunities for compiler optimization. The limitation can be compensated by introducing more hash-mapped shadow registers. Figure 8 shows the speedup with different shadow register architectures. For the shadow registers with multiple control bits (*Sreg*) as proposed in [8], three registers are used due to the control bit cost. With three hash-mapped shadow registers (*Hsreg*) using one control bit, only one case, the *blowfishd* with 3-input constraint, degrades its performance. Other benchmarks achieve comparable performance thanks to our global shadow register allocation algorithm. By providing more hash-mapped shadow registers yet still use one control bit, we can further improve the performance. The results shown in Figure 8 indicate that we can almost reach ideal speedup (98% of the performance gap is closed) by providing five and eight hash-mapped shadow registers for the 3-input and 4-input constraints respectively. However, three shadow registers with multiple control bits [8] still leaves 30% performance gap on average for the two constraints.

We further compare our proposed shadow registers with partial register replication (*Rreg*) method used in [10]. Figure 9 shows the speedup under different input constraints and different number of registers. With the same number of registers, shadow register architecture consistently outperforms partial register replication, which leaves a 49% and 46% performance gap open for 3-input and 4-input constraints even with eight replicated registers.

To examine the impact of the hash function, we also compare our generated hash function with a simple mod function (i.e., i mod K). In our experiment, we find that it can achieve comparable results for most cases. However, it will demand more shadow registers to achieve the same speedup. For instance, it needs four and three more shadow registers to achieve the same speedup for the *adpcmc* and *blowfishe* with 3-input constraint.

5. CONCLUSIONS

The data bandwidth problem is seriously limiting the performance of application-specific instruction set processors. In this paper we propose a new low-cost architectural extension, which employs the hash-mapped shadow registers to directly address the data bandwidth limitation. We also present a simultaneous global register binding with a hash function generation algorithm to fully exploit the benefits of hash-mapped shadow registers across the basic block boundaries. The application of our approach results in a very promising performance improvement.

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] C. J. Alpert and A. B. Kahng, "Multi-Way Partitioning Via Spacefilling Curves and Dynamic Programming," in *Proc. 31st Design Automation* Conference, pp. 652-657, Jun. 1994.

[3] C. J. Alpert and A. B. Kahng, "Recent Developments in Netlist Partitioning: A Survey," *Integration: the VLSI Journal*, vol. 19(1-2), pp. 1-81, 1995.

[4] 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.

[5] D. Burger, T. Austin, and S. Bennett, "Evaluating Future Microprocessors: The SimpleScalar Toolset," *Technical Report, CS-TR96-1308*, Univ. of Wisconsin - Madison, 1996.
[6] N. Clark, H. Zhong, and S. Mahlke, "Processor Acceleration Through Automated Instruction Set Customization," in *Proc. International Symposium on Microarchitecture*, pp. 129-140, Dec. 2003.
[7] 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.
[8] J. Cong, et al., "Instruction Set Extension for Configurable Professors with Shadow Registers," in *Proc. ACM International Symposium on Field-Programmable Gate Arrays,* Feb. 2005.
[9] M. R. Guthaus, et al., "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *IEEE 4th Workshop on Workload Characterization*, Dec. 2001.
[10] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, "The Chimaera Reconfigurable Functional Unit," *IEEE Transactions on VLSI Systems*, vol. 12(2), pp. 206-217, Feb. 2004.
[11] 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.
[12] 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.
[13] 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.
[14] M. Poletto and V. Sarkar, "Linear Scan Register Allocation," *ACM Transactions on Programming Languages and Systems*, vol. 21(5), pp. 895-913, Sep. 1999.
[15] S. Rixner, et al., "Register Organization for Media Processing," in *Proc. Sixth International Symposium on High-Performance Computer Architecture*, pp. 375-386, Jan. 2000.
[16] 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.
[17] Altera Corp., http://www.altera.com.
[18] Tensilica Inc., http://www.tensilica.com.
[19] Xilinx Inc., http://www.xilinx.com.

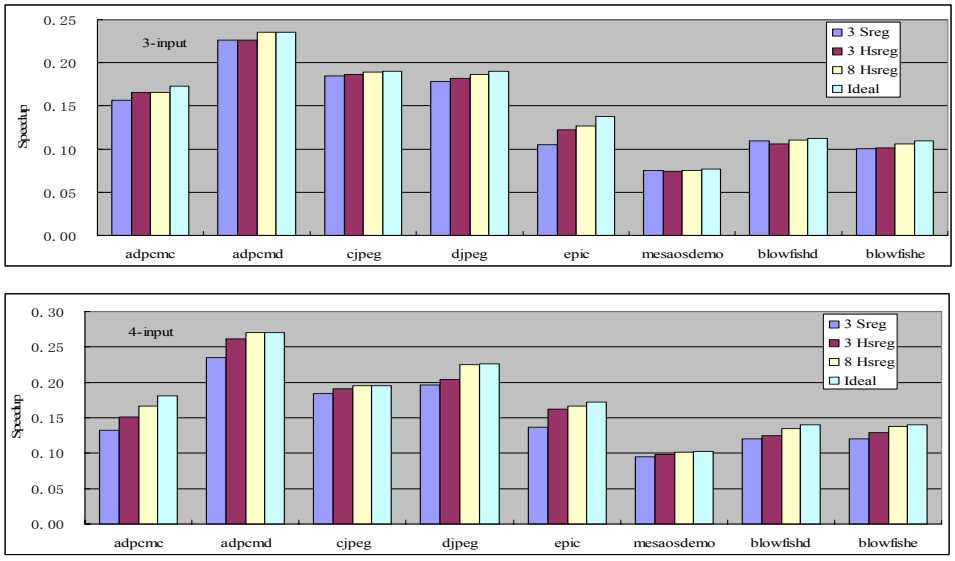

Figure 8. Speedup under different shadow register architectures.

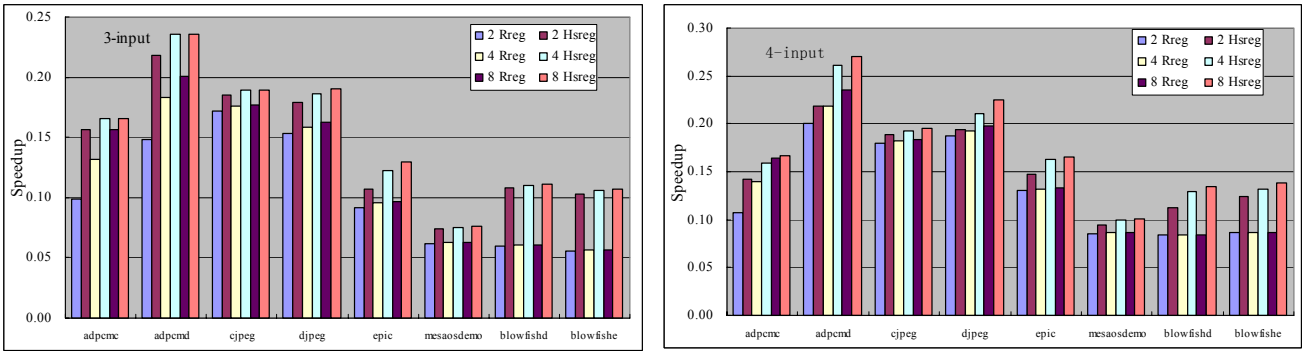

Figure 9. Speedup under different number of shadow registers and duplication registers.